Formalizing Style to Understand Descriptions of Software Architecture

Gregory Abowd Robert Allen David Garlan

January 1995
CMU-CS-95-111

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Gregory Abowd's current address is College of Computing, Georgia Institute of Technology, Atlanta, Georgia. This paper extends work published as "Using Style to Understand Descriptions of Software Architecture," *Proc. SIGSOFT'93: Foundations of Software Engineering*, December 1993.

Abstract

The software architecture of most systems is usually described informally and diagrammatically by means of boxes and lines. In order for these descriptions to be meaningful, the diagrams are understood by interpreting the boxes and lines in specific, conventionalized ways. The informal, imprecise nature of these interpretations has a number of limitations. In this paper we consider these conventionalized interpretations as architectural styles and provide a formal framework for their uniform definition. In addition to providing a template for precisely defining new architectural styles, this framework allows for analysis within and between different architectural styles.

The research reported here was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330; by National Science Foundation Grant CCR-9109469; and by a grant from Siemens Corporate Research. 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 US Department of Defense, the United States Government, the National Science Foundation, or Siemens Corporation. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

Keywords: software architecture, software design, architectural style, architectural description, architectural analysis, formal specification, the Z notation

1 Introduction

Software architecture is an important level of description for software systems [18, 30]. At this level of abstraction key design issues include gross-level decomposition of a system into interacting subsystems, the assignment of function to computational components, protocols of interaction between those components, global system properties (such as throughput and latency), and life-cycle issues (such as maintainability, extent of reuse, and platform independence).

When designers discuss or present a software architecture for a specific system, they typically treat the system as a collection of interacting *components*. Components define the primary computations of the application. The interactions, or *connections*, between components define the ways in which the components communicate, or otherwise interact with each other. In practice a large variety of component and connector types are used to represent different forms of computation or interaction [18]. Examples of component types include filters, objects, databases, and servers. Examples of connector types include pipes, procedure calls, message passing, and event broadcast.

Most architectural descriptions are informal and diagrammatic, using annotated boxes to represent components and lines to represent the connections. In order for these descriptions to be meaningful at all, a number of questions about the described system must be answered:

- What computations do the boxes and their annotations represent?
- Are the boxes somehow similar in behavior?
- What control/data relationships are indicated by the lines?
- How is the overall behavior of the system determined by the behavior of its parts?
- Does the diagram make sense—that is, does it represent a legal configuration of boxes and lines?

Simple box-and-line diagrams by themselves cannot answers these questions directly. Consequently designers typically resort to conventional interpretations of their diagrams in order to provide those answers. For example, an architectural description for one system might use boxes to represent filters and lines to represent piped (dataflow) channels between filters. For another system, boxes might represent abstract data types or objects, and lines might represent procedure calls. In a system description containing more than one kind of component or connection type, the different types are often distinguished by different graphical conventions.

While useful in documenting system designs, such diagrams—even with their conventional interpretations—have a number of serious limitations. Because they are imprecise, it is difficult or even impossible to attach unambiguous meanings to the descriptions. This makes it difficult to know when an implementation agrees with the architectural description. Consequently, it is difficult to know how changes to one affects the other. Similarly, lack of precision precludes formal analysis: it is virtually impossible to reason formally about a system's architectural description or to make rigorous comparisons between different architectural descriptions.

The most common solution to the inadequacies of informal interpretation of architectural description is to constrain the architectural notation so that it maps directly into a well-defined execution model. This is typically done by casting architectural descriptions in terms of module-level notations provided by programming languages (e.g., Ada packages and tasks, C++ classes, etc.). For example, components can be restricted to be abstract data types whose interface is described solely in terms procedure signatures, and connectors can be restricted to procedure call. When constrained in this way, architectural descriptions can be mapped directly to facilities of a programming language (or other executable base), and can thereby be given precise meanings.[1]

This approach, however, has a number of problems. Most significantly, it limits the expressiveness of architectural description to just those structures and building blocks supported directly by the target

[1] This assumes, of course, that the programming language itself has well-defined semantics.

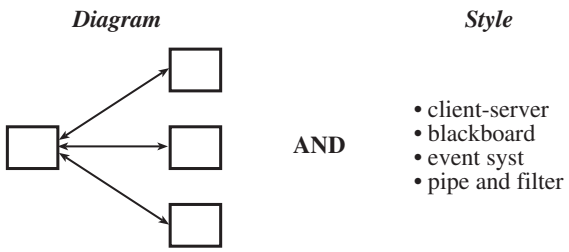

Figure 1: How style distinguishes similar descriptions

implementation language. If, for instance, architectural connections have to be phrased in terms of procedure calls, then alternative forms of interaction—such as event broadcast or higher-level interactions characterized by protocols of communication—cannot be represented directly. Moreover, alternative forms of interaction must be encoded into the primitives at hand, obscuring the intent of the designer. Finally, the relatively low level of description may make it difficult to understand and reason about the architectural design.

In this paper we advocate a different approach: permit a variety of conventional interpretations to be assigned to architectural diagrams, but create a framework for understanding and defining them more precisely. To make this possible what is needed is a flexible way to assign formal semantics to architectural descriptions in a way that is consistent with the informal conventions used by their creators. In this way, designers can use the abstractions that are appropriate to the architectural description at hand, but still have the precision of a formal model.

More specifically, as we will see, we can view the collection of conventions that are used to interpret a class of architectural descriptions as defining an *architectural style*. To understand the meaning of a specific architectural design then requires both a description of the design (usually in the form of an architectural diagram), as well as an indication of the style under which the description is to be understood.

To elaborate, as illustrated in Figure 1, an architectural diagram identifies the number and connectivity of computational entities. However, it is the style that tells us what kinds of components should exist, the control/data relationships between components, and other semantic details, such as constraints on topology. For example, if we interpret the diagram of Figure 1 with respect to a client-server architectural style, the system could be understood as consisting of two kinds of components (clients and servers) connected by a request-reply protocol initiated by the clients. Interpreting the diagram as a blackboard system [20], would indicate the presence of a central blackboard together with three knowledge sources. Interpreting the same diagram under the pipe-filter style, on the other hand, could allow us to infer that the diagram is illegal, if pipes are not used for two-way communication between components in that style.

In this paper, we show that this basic idea can be made precise. Specifically, architectural styles can be described formally in terms of a small set of mappings from the syntactic domain of architectural descriptions to the semantic domain of architectural meaning. (See Figure 2.) The approach provides a framework in which new styles can be defined by instantiating similar sets of definitions. The formal model further makes it possible to gain insight into the properties of a style and its relationships to other styles.

The main thrust of our argument and examples is to demonstrate how to give meanings to architectural descriptions. In one respect this is nothing new: programming language researchers have been providing denotational semantics of programming languages for years. What *is* novel, however, is the specialization of the general semantic approach to the problem of understanding software architecture. As we will show, this can be done by providing a syntactic and semantic framework in which architectural styles give meanings to architectural diagrams.

The specialization of general theory to this particular domain has a number of significant engineering benefits. First, it provides a template for formalizing new architectural styles in a uniform way, thereby simplifying and regularizing the way styles are given meanings. Second, it provides uniform criteria (in

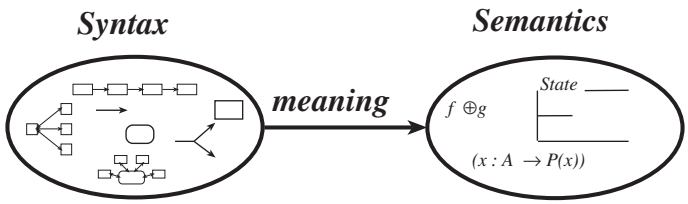

Figure 2: Approach to formalizing architectural style

the form of proof obligations) for demonstrating that the notational constraints on a style are sufficient to provide meanings for all described systems. Third, it makes possible a unified semantic base through which different stylistic interpretations can be compared.

1.1 Related work

The ideas presented in this paper are most closely related to four other areas of research: architectural taxonomies and handbooks; languages for architectural description; work on domain-specific software architectures; and other formal models for architectural specification.

Architectural taxonomies and handbooks

Architectural design has long been recognized as a critical aspect in engineering large software systems [9, 29]. However, it is only recently that software architecture has begun to emerge as discipline of study in its own right. This has come about in part by a recognition of the central role of common design patterns and idioms — or architectural style. Among early efforts to identify, name, and analyze these patterns, in 1989 Shaw categorized a number of idioms [33] and later Garlan and Shaw [18] extended this list, providing several examples of their use in understanding real systems. Concurrently, Perry and Wolf [30] also recognized the importance of architectural patterns and outlined the use of styles in characterizing applications such as compilers.

A different, but related, area of activity has recently emerged in the object-oriented community through the articulation of (object-oriented) design patterns [10]. Inspired, in part, by Christopher Alexander's work on pattern languages [1], these efforts have led to handbooks of common patterns for organizing software [11, 31]. The patterns usually consist of a small number of objects that interact in specific ways. (A typical example is the Model-View-Controller pattern, identified by the creators of Smalltalk for organizing user interface software [23].)

Our research builds on earlier taxonomic efforts by recognizing the importance of architectural abstractions as semantic entities worthy of study. But it goes beyond that work by showing how to make architectural descriptions more precise. In particular, early work on cataloging the ways software engineers express their architectural designs, convinced us that it is foolhardy to attempt to limit the number of architectural patterns, or simply to reduce them to more familiar, primitive programming language constructs. On the other hand, it is clear that the study of architectural patterns can benefit substantially from techniques for making their description more formal, and hence more analyzable [35].

Languages for architectural description

In an attempt to provide architectural design with better notations, several new languages have been proposed for architectural description. Rapide [24] provides a module description language, whose interface model is based on SML [26] augmented with events and event patterns. Unicon [34] provides an architectural description language in which both components and connectors have interfaces and can be associated

with implementations. (In the case of a Unicon connector, the "implementation" is called a "protocol.") Wright [4, 5] is an architectural specification language that allows one to define the semantics of connectors as formal protocols in a variant of CSP [21]. To the extent that these languages have well-defined semantics, they provide a formal basis for architectural description. However, in their current form, none of them is specifically concerned with the definition of architectural style: while stylistic constraints can be added to the description of a specific system, unlike our work, none of the languages currently make it possible to define an architectural style as an independent semantic entity.

Closer in spirit to our work is the architectural description framework provided by the Aesop System [12]. Aesop was specifically designed to support the definition of new architectural styles. New styles are defined a system of object types, which provide a design vocabulary for the style, and are then used to support a design environment specialized for the style. Stylistic constraints are enforced by the "methods" of the object types. As such, Aesop provides an operational basis for style definition. This contrasts with the more direct "denotational" approach of this paper.

Domain-specific software architectures

A growing number of industrial research and development efforts are creating domain-specific architectural styles—or "reference architectures"— for specific product families [7, 8, 25]. This work is based on the idea that a common architecture of a collection of related systems can be extracted so that each new system can be built by "instantiating" the shared architecture. Examples include the standard decomposition of a compiler (which permits undergraduates to construct a new compiler in a semester), standardized communication protocols (which allow vendors to interoperate by providing services a different layers of abstraction), fourth generation languages (which exploit the common patterns of business information processing), user interface toolkits and frameworks, and various product architectures in domains such as command and control, avionics, manufacturing, and mobile robotics [19, 38].

Our work was inspired, in part, by the demonstrable benefits in developing such styles. However, to the extent that those efforts have formalized their architectural frameworks at all, the semantic descriptions are developed from scratch, and each uses different, idiosyncratic conventions and semantic bases. Such formal descriptions are therefore difficult to develop and, having developed them, few comparisons can be made between different styles. In contrast, our work attempts to find a common basis for defining many architectural styles and for making semantic comparisons between them.

Other formal models for architecture

In earlier work the authors and other colleagues, have provided formal models for several specific architectural styles, including a class of signal processing systems [14], a pipe-filter style [2], and an implicit invocation style [16]. Each of these specifications was an independent specification effort, and required considerable expertise. This previous experience provided strong motivation for developing a unified framework for defining architectural styles. In fact, two examples of architectural styles used in this paper were adapted from our earlier work.

Other bases for formal modelling of architecture have been proposed. In their investigations of architectural refinement, Moriconi and his colleagues have experimented both first order predicate logic and with Lamport's Temporal Logic of Actions [27, 28]. In Moriconi's work a style is taken to be a kind of theory in first order predicate logic. While that view of architectural style is consistent with ours, in this paper we show how to provide structure to the formal description of architectural style, and thereby simplify and regularize the definition of the theory associated with it.

Inverardi and Wolf have investigated the Chemical Abstract Machine [6] as a formal basis for architectural description [22]. Architectural elements are represented by "molecules" and architectural interaction by "reactions." It remains to be seen whether this provides a better formal basis than the (set-theoretic) one we have chosen. In any case, to date their work has primarily focused on the description of specific architectures, rather than architectural styles.

Finally, as noted earlier, two of the authors (Allen and Garlan) have developed an alternative formalism for architectural specification based on CSP [5]. While the use of CSP has a number of benefits over Z—especially for describing the dynamic behavior of a system—thus far, their architectural specification language does not support the definition of architectural styles.

1.2 Overview of the rest of the paper

In Section 2, we begin by outlining the method we use to define an architectural style. In Section 3, we abstract and formalize the concepts behind the box-and-line diagrams that are prevalent in current informal architectural descriptions as a *syntactic domain*. We then demonstrate for two particular architectural styles the definition of a semantic model to describe the overall behavior of a system and show how architectural syntax can be mapped into that model as a formal style definition. We define a pipe-filter style in Section 4 and an implicit invocation, or event system, style in Section 5. Finally, in Section 6 we show how these semantic underpinnings support the analysis and comparison of styles.

Throughout the paper, we use the Z specification language to describe the formal model [36]. Appendix A summarizes the Z notation used in this paper. However, it is important to note that the use of Z is not critical to the approach that we are advocating. Indeed, the main contribution of this paper is in defining the *framework* for style definition and then demonstrating its value for architectural analysis of various styles. Many other formal notations would have sufficed for this purpose.

2 What's in a Style?

In order to provide a precise meaning for architectural descriptions it is important to distinguish the abstract syntactic domain of architectural descriptions from the semantic domain of architectural meanings. Having done this we can then provide a map, or meaning function, from one to the other.

We take as our starting point the view that the syntactic domain of architectural description (among other things) supports the description of systems in terms of three basic syntactic classes: *components*, which are the locus of computation; *connectors*, which define the interactions between components;[2] and *configurations*, which are collections of interacting components and connectors. Additionally, various style-specific concrete notations may be used to represent these visually, facilitate the description of legal computations and interactions, and constrain the set of describable systems. We are not as concerned in this paper with the specifics of these concrete notations as we are with their purpose in easing the description of architectural instances.

A purely syntactic description may have some benefits as an informal design notation. For example, the connectors may be interpreted as defining data flows through the system. But as we argued in the introduction, such informal approaches have serious limitations. In particular, questions such as how components compute, what data is communicated, or how the flow of information is controlled, cannot be answered with any precision. Since it is the purpose of this paper to provide an improved basis for understanding the meaning of architectural descriptions, we will take the view that architectural style is an interpretation from syntax to semantics (see Figure 2), and outline a framework for precise style definition.

In this framework, style definition starts with a formal definition of the syntactic domain in which architectures are described. In Section 3 we do this generically by providing formal definitions of the syntactic classes: component, connector and configuration. These represent the basic elements of an architectural diagram. Next, for each style we must define a semantic model that captures both the static and dynamic meanings of the class of systems built in that style. Finally, as with a denotational approach to programming languages, we provide a mapping from the syntactic descriptions to the semantic model for the style. Given the nature of architectural descriptions, this amounts to the definition of three *meaning functions* that link

[2] In practice the implementation of a connector may involve some computations – such as buffers for communication, dispatching of events, synchronization over shared variables, etc. But we distinguish these computations from those of the components: the former is typically a computation of the underlying support system, while the latter is a computation of the application itself.

the syntactic descriptions to their semantic counterparts. For a style X, we would declare the meaning functions as partial functions from the abstract syntax to the semantic models.

$$\mathcal{M}^X_{Comp} : Component \pfun Comp^X_{sem}$$
$$\mathcal{M}^X_{Conn} : Connector \pfun Conn^X_{sem}$$
$$\mathcal{M}^X_{Conf} : Configuration \pfun Conf^X_{sem}$$

Here *Component* is the abstract syntactic class of components (to be defined in Section 3) and $Comp^X_{sem}$ denotes the semantic model of a component in style X. Thus, $\mathcal{M}^X_{Comp}$ is a meaning function from the general abstract syntax for components to the style-specific semantic model. It is modeled as a partial function (using the Z symbol $\pfun$) to indicate that some elements of *Component* may not have a meaning in a given style. In fact, as we will see, part of the definition of a style will be to determine which syntactic elements can legally be assigned a meaning. This is done by defining explicitly the domain of the meaning functions—or using the generic notation above, $\mathrm{dom}(\mathcal{M}^X_{Comp})$. Similar conventions are used to define the meaning functions for connectors and configurations.

The final step in the formal definition of an architectural style is to make explicit the constraints that this style imposes on the syntactic descriptions. Because the meaning functions are declared as partial functions on the syntactic domains, not every syntactic construct may have a meaning in a given style. Expressing these constraints explicitly generates a proof obligation to show that the meaning function is well-defined for all syntactic elements that meet the constraints. By making the constraints explicit we define precisely the descriptions that are reasonable in the style.

A formal definition of an architectural style, based on the method outlined above, provides a foundation for further analysis of the style. We discuss two different forms of analysis in this paper. The first form of analysis is *within* a particular style, identifying important substyles that can be understood as further syntactic restrictions on a more general style. The second form of analysis is *between* styles, comparing different semantic models to see if they share similar properties.

To summarize, the steps we will follow are:

1. formalize abstract syntax for architectures
2. for a given style:
   - define the semantic model
   - discuss concrete syntax for easing syntactic descriptions in a given style
   - define the mapping from abstract syntax into semantic model
   - make explicit the constraints on the syntax
3. demonstrate analysis within and between formally defined architectural styles.

3 The Abstract Syntax of Software Architectures

The basic syntactic elements of an architectural description are *components*, *connectors*, and *configurations* of components and connectors. In this section we formalize these elements.

3.1 Components

Components are the active, computational entities of a system (see Figure 3). They accomplish tasks through internal computation and external communication with the rest of the system. The relationship between a component and its environment is defined explicitly as a collection of interaction points, or *ports*. Intuitively, ports generalize the traditional notion of a module interface. In the simplest case, a port might represent a procedure that can be called or a variable that can be accessed in an interaction with another component.

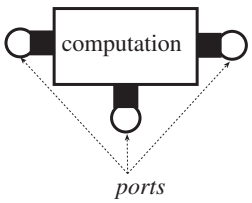

Figure 3: A component and a connector

But a port might also represent something much more complex, such as a collection of procedures, a set of events that can be broadcast, or a database access protocol. (For more details on the use of ports for defining complex interfaces see [4].)

We differentiate between components with the same port interface based on a description of the computation they perform. In this abstraction of component syntax, we model this reference to computational behavior with a placeholder for some concrete computational description. That is to say, we leave the details of port naming and description unbound at this point. Since we are not concerned with details of the construction of ports or the computational description for components, we model these as "given" sets.[3] An architectural component, as a syntactic entity, is a collection of ports together with a description of its computation. We use the Z schema, *Component* to represent this

$[PORT, COMPDESC]$

Component

$ports : \mathbb{P}\ PORT$

$description : COMPDESC$

3.2 Connectors

Connectors define the interaction between components (see Figure 4). Each connector provides a way for a

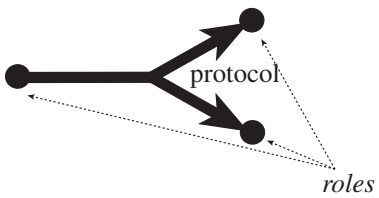

Figure 4: A connector

collection of ports to come into contact, and logically defines the protocol through which a set of components will interact.

Like components, connectors are defined as independent entities. A connector has an interface that consists of a set of *roles*. Each role defines the expected behavior of one of the participants in an interaction. For example, a pipe would have a reader and a writer role; a multicast connector would have a single

[3] In Z a *given set* simply defines a primitive collection of elements, which can be compared for equality, but otherwise have no internal structure—see Appendix A.

announcer and multiple receiver roles; a client-server connector would have requester and provider roles. The overall behavior of a connector (and hence of the interaction it specifies) is logically defined by a protocol. For example, the protocol of a client-server connector might require that initialization occur before any request is made. Thus the description of the protocol of interaction provided by a connector is separated from its interface (of roles) in the same way that the computation description of a component is separated from its interface (of ports).

Again, in this model we are not concerned with the detailed specification of roles and protocol of interaction, so we introduce these notions as given sets. An architectural connector is then modeled as a collection of roles and a description of its interaction protocol, as defined in the schema *Connector*.

$[ROLE, CONNDESC]$

Connector

$roles : \mathbb{P}\ ROLE$
$description : CONNDESC$

3.3 Configurations

A configuration is a collection of component instances which interact by means of connector instances (see Figure 5). Instances of components and connectors are identified by naming elements from the syntactic

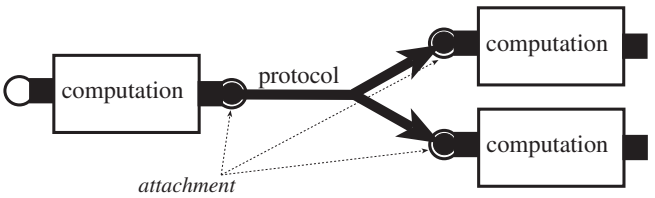

Figure 5: A configuration

class. To name instances of components and connectors we introduce two new given sets, *COMPNAME* and *CONNNAME*. These sets are also used to name instances of ports or roles (resp.) associated with a component or connector (resp.), and so we introduce two type synonyms for convenience.

$[COMPNAME, CONNNAME]$
$PortInst == COMPNAME \times PORT$
$RoleInst == CONNNAME \times ROLE$

The association between component and connector instances is modeled by an *attachment* between the roles of the connectors and the ports of the components. This reflects the intuition discussed above, in which the connector interface identifies roles in the interaction that are to be filled by various component ports. This leads to a certain asymmetry: while a port may fill many roles, meeting the needs of several different communications, a role may have at most one port that fills it.

The model for a configuration is given below. Instances of components and connectors are modeled by partial functions from the naming set to the syntactic class. Each name-element pair in these functions indicates an instance of that element in the configuration. Attachments are modeled as a partial function from the roles of the connector instances to the ports of the component instances. Using a partial function enforces the consistency constraints described above: namely, that there is at most one port for each role, but possibly multiple roles for a single port.

Configuration

$components : COMPNAME \pfun Component$
$connectors : CONNNAME \pfun Connector$
$attachment : RoleInst \pfun PortInst$

$\forall cn : CONNNAME; r : ROLE \mid (cn, r) \in \mathrm{dom}\, attachment$
$\quad \bullet\ cn \in \mathrm{dom}\, connectors \land r \in (connectors(cn)).roles$

$\forall cn : COMPNAME; p : PORT \mid (cn, p) \in \mathrm{ran}\, attachment$
$\quad \bullet\ cn \in \mathrm{dom}\, components \land p \in (components(cn)).ports$

The schema *Configuration* imposes two additional constraints (below the separating line) that must be satisfied by all configurations. The first constraint ensures that any role instance in the *attachment* is a role for some named connector in the configuration. The second constraint similarly ensures that all port instances described by the configuration appear on an actual component instance. Together, these two constraints enforce a lexical scoping on attachments within a configuration.

4 The Pipe-Filter Style

In this section, we show how this framework can be used to model the syntactic elements of a pipe-filter style (PF). This style is representative of coarse-grained dataflow systems such as those supported by Unix pipes. Figure 6 provides a pictorial overview of the pipe-filter architectural style. Components are filters. Their computation is a *transition* function from *input* ports to *output* ports. The connectors, *pipes*, support dataflow between two filters.

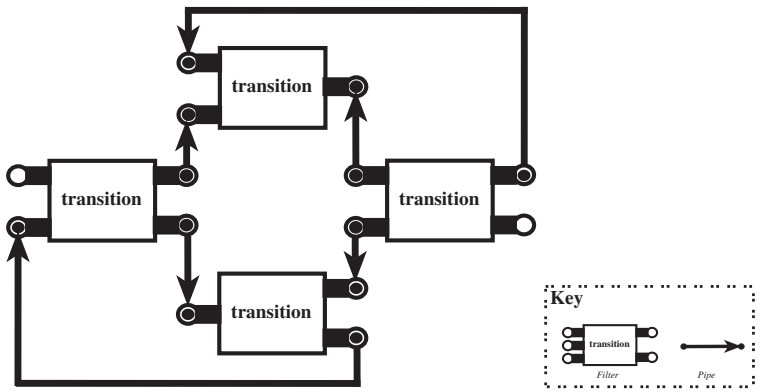

Figure 6: An instance of the pipe-filter style

4.1 Semantic Model

The first part of defining a style is to provide a semantic model for the components, connectors, and configurations of the style. This is perhaps the hardest part of the process, since to do this properly we must come to grips with the intuition behind the use of the style. In the case of PF, an appropriate formal description of the semantic domain already exists [2, 3]. Here we will use only those aspects of the model that are necessary to illustrate the basic ideas.

The PF style interprets components as filters, which are typed stream transducers. These can be modeled as state machines that receive their input and place their output as sequences on data ports. We do not

wish to uncover the details of how the internal state and data are described, so we declare them as given sets in our specification. Data ports define the interfaces for filters and we also introduce them as a given set in our model. Note that these are distinct from the ports that form the interface for components in the syntactic descriptions.

$[STATE, DATA, DATAPORT]$

In order to define the behavior of a filter, we must know its input and output data ports and the type of data that may be passed along each data port. This latter information can be represented by a (partial) function from data ports to their alphabet. At any point in time, the data ports of the filter will hold all data (as a sequence) that has been received (for input data ports) or produced (for output data ports) but not yet removed. The state machine behavior of the filter is modeled as a transition function that takes an internal state and input data and results in a new internal state and output data. In addition we identify a starting internal state. This information about a filter is formalized in the schema *Filter*. Some constraints on filters that we enforce are:

- input and output data ports are distinct (first predicate);
- a filter transition is determined by looking at data on the input ports only and results in information provided to the output ports only (the final predicate).

Filter

$$
\begin{array}{l}
inputs, outputs : \mathbb{P}\, DATAPORT \\
alphabet : DATAPORT \pfun \mathbb{P}\, DATA \\
states : \mathbb{P}\, STATE \\
start : STATE \\
transitions : (STATE \times (DATAPORT \pfun \operatorname{seq} DATA)) \\
\qquad \rightarrow (STATE \times (DATAPORT \pfun \operatorname{seq} DATA)) \\
\hline
inputs \cap outputs = \varnothing \\
\operatorname{dom} alphabet = inputs \cup outputs \\
start \in states \\
\\
\forall\, s_1, s_2 : STATE;\ ps_1, ps_2 : DATAPORT \pfun \operatorname{seq} DATA \\
\quad \bullet\ ((s_1, ps_1), (s_2, ps_2)) \in transitions \Rightarrow \\
\qquad\qquad s_1 \in states \land s_2 \in states \\
\qquad \land \quad \operatorname{dom} ps_1 = inputs \land \operatorname{dom} ps_2 = outputs \\
\qquad \land \quad (\forall\, i : inputs \bullet \operatorname{ran}(ps_1(i)) \subseteq alphabet(i)) \\
\qquad \land \quad (\forall\, o : outputs \bullet \operatorname{ran}(ps_2(o)) \subseteq alphabet(o))
\end{array}
$$

We define the semantics of a filter operationally. At any point in a computation, a filter is defined by its current internal state, constrained to be in the set of possible states for the filter, and the data at each of its input and output ports (which must be in the alphabet of that port).

FilterState

$$
\begin{array}{l}
f : Filter \\
curstate : STATE \\
instate, outstate : DATAPORT \pfun \operatorname{seq} DATA \\
\hline
curstate \in f.states \\
\operatorname{dom} instate = f.inputs \\
\operatorname{dom} outstate = f.outputs \\
\forall\, p : f.inputs \bullet \operatorname{ran}(instate(p)) \subseteq f.alphabet(p) \\
\forall\, p : f.outputs \bullet \operatorname{ran}(outstate(p)) \subseteq f.alphabet(p)
\end{array}
$$

A single computational step for a filter transforms some input data into output data. The order of data is preserved, so input data is consumed in the order it arrived and output data is kept in the order it is produced. The result of a computation step for a filter is the removal of some data off the input ports, a transformation of that data, which will depend on the filter's current internal state, a change in the current state and the addition of the transformed data to the output ports. The schema *FilterCompute* encapsulates just such a computational step. We make use of the Δ convention to describe this transition from one state of the filter to another (see Appendix A).

FilterStep

$$\Delta FilterState$$

$$f' = f$$

$$\exists\, in, out : DATAPORT \pfun \operatorname{seq} DATA \bullet$$
$$((curstate, in), (curstate', out)) \in f.transitions$$
$$\land\ \forall\, p : f.inputs \bullet instate(p) = in(p) \frown instate'(p)$$
$$\land\ \forall\, p : f.outputs \bullet outstate'(p) = outstate(p) \frown out(p)$$

The data ports of filters are connected by pipes, which we model as typed streams of data. Each pipe has a distinct source and sink for receiving and sending data. Recall that a *DATAPORT* represents an input or an output of some particular filter. Thus, a pipe represents a data transmission from one filter to another.

Pipe

$$source, sink : DATAPORT$$
$$alphabet : \mathbb{P}\, DATA$$

$$source \neq sink$$

The protocol or behavior of a pipe is defined by giving its transmission policy. At any point in time, the pipe has some data residing at its source port and some data at its sink port.

PipeState

$$p : Pipe$$
$$sourcedata : \operatorname{seq} DATA$$
$$sinkdata : \operatorname{seq} DATA$$

$$\operatorname{ran} sourcedata \subseteq p.alphabet$$
$$\operatorname{ran} sinkdata \subseteq p.alphabet$$

A single step in the behavior of a pipe results in some nonempty subsequence of data being removed from the source data port, in the order in which it arrived there, and being delivered, unchanged in content and order, to the sink data port.

PipeStep

$$\Delta PipeState$$

$$p = p'$$

$$\exists\, deliver : \operatorname{seq} DATA \mid \#deliver > 0 \bullet$$
$$(deliver \frown sourcedata' = sourcedata) \land (sinkdata' = sinkdata \frown deliver)$$

We can now model a pipe-filter configuration as a set of filters connected by pipes. Because the *DATAPORT* identifiers represent global names, we disallow name clashes between the data ports of distinct filters and pipes. The set of interactions in the system is modeled by identifying each pipe *source* with a unique filter output and each pipe *sink* with a unique filter input.

```
inputs: char in;
outputs: char out;
execution:
  char c;
  while (TRUE) {
    c = read(in);
    if (c >= 'a' && c <= 'z') {write(out,c+'A'-'a');}
    else {write(out,c);}
  }
```

Figure 7: Concrete Description of a Capitalizing Filter

InteractingFilterSet

$$
\begin{array}{l}
\mathit{filters} : \mathbb{P}\ \mathit{Filter} \\
\mathit{pipes} : \mathbb{P}\ \mathit{Pipe} \\
\hline
\forall f_1, f_2 : \mathit{filters} \mid f_1 \neq f_2 \bullet (f_1.\mathit{inputs} \cup f_1.\mathit{outputs}) \cap (f_2.\mathit{inputs} \cup f_2.\mathit{outputs}) = \varnothing \\
\forall p_1, p_2 : \mathit{pipes} \mid p_1 \neq p_2 \bullet \{p_1.\mathit{source}, p_1.\mathit{sink}\} \cap \{p_2.\mathit{source}, p_2.\mathit{sink}\} = \varnothing \\
\forall p : \mathit{pipes} \bullet \exists f_1, f_2 : \mathit{filters} \bullet \\
\qquad p.\mathit{source} \in f_1.\mathit{outputs} \\
\quad \wedge \quad p.\mathit{sink} \in f_2.\mathit{inputs} \\
\quad \wedge \quad f_1.\mathit{alphabet}(p.\mathit{source}) = p.\mathit{alphabet} \\
\quad \wedge \quad f_2.\mathit{alphabet}(p.\mathit{sink}) = p.\mathit{alphabet}
\end{array}
$$

The behavior of an interacting set of filters is defined as the behaviors of the constituent filters and pipes. A step in this behavior is either a computation step for one filter or a transmission step for one pipe, all else remaining unchanged. Details of this behavioral specification have been omitted here but can be found elsewhere [3].

4.2 Concrete Syntax

The second part of a style definition is the creation of a style-specific concrete syntax. While the details of such syntax are important, in this paper we are more concerned with understanding the relationship between these descriptions and their associated meanings. In that regard, it is enough to know that there exist filter and pipe description languages that determine the interesting subset of the possible component and connector descriptions in the PF style. Formally, we represent these languages as subsets of the respective description languages introduced in Section 3.

$$
\begin{array}{l}
\mathit{FilterDescriptions} : \mathbb{P}\ \mathit{COMPDESC} \\
\mathit{PipeDescriptions} : \mathbb{P}\ \mathit{CONNDESC}
\end{array}
$$

For concreteness, Figure 7 illustrates the definition of a filter that capitalizes its character input stream using one notation developed for this style [3].

4.3 Meaning Functions

The third part of a style description is to define the meaning of the architectural syntax in terms of the semantic model.

As indicated in Section 2, to give meaning to components we need to specify a partial function of the form:

$$\mathcal{M}^X_{Comp} : Component \pfun Comp^X_{sem}$$

From the definition of *Filter*, we can see that it is possible for two filters to be identical up to naming of data ports and states. Therefore, we can define an equivalence relation on elements in *Filter*. We treat two filters as equivalent if and only if there is an isomorphism between their states, and their input and output data ports that preserves the behavior defined by their transition functions. This equivalence relation is denoted by $\equiv_{fil}$. The detailed definition of $\equiv_{fil}$ is not given below, though it is straightforward.

$$_ \equiv_{fil} _ : Filter \leftrightarrow Filter$$

The meaning function for PF components, written below as $\mathcal{M}^{PF}_{Comp}$, identifies the syntactic element *Component* with an equivalence class of filters. So in this example, $Comp^X_{sem}$ is replaced by sets of filters, or $\mathbb{P}\ Filter$.

To complete the mapping from syntax to semantics, we need to have an injective function, called *DataPort* below, from named instances of the syntactic ports to the semantic data ports. The reason we have the function *DataPorts* is to provide a way of distinguishing aspects of the semantic model that are named in the syntactic descriptions. The function $\mathcal{M}^{PF}_{Comp}$ provides a correspondence between the description and the semantic model. The syntax, however, provides a means of naming parts, or aspects, of a computation. In the case of PF, different inputs and different outputs are distinguished. It is therefore necessary to carry that distinction into the semantic model.

For example, a filter might divide its input into two output streams depending on the values seen (*e.g.* all values less than a threshold go to one, and all above it to another). We need to be able to specify which pipes in a system get which output ports. If the high values go to the handler for low values, and vice-versa, the system would have a dramatically different effect.

As we will see when the entire system is defined, *DataPort* serves to ensure that the correct interactions are indeed achieved. It will also allow multiple instances of the same filter to be used in a system, by mapping the local names of the syntactic description into the global names of the semantic model.

$$\begin{array}{l} DataPort : PortInst \rightarrowtail DATAPORT \\ \mathcal{M}^{PF}_{Comp} : Component \pfun \mathbb{P}\ Filter \\ \hline \forall\, c : Component; f_1, f_2 : Filter \mid f_1 \in \mathcal{M}^{PF}_{Comp}(c) \\ \quad \bullet\ f_2 \in \mathcal{M}^{PF}_{Comp}(c) \Leftrightarrow f_1 \equiv_{fil} f_2 \\ \forall\, c : Component; n : COMPNAME \mid c \in \mathrm{dom}\, \mathcal{M}^{PF}_{Comp} \\ \quad \bullet\ \exists f : \mathcal{M}^{PF}_{Comp}(c) \bullet DataPort(\!|\{n\} \times c.ports|\!) = (f.inputs \cup f.outputs) \end{array}$$

In Section 4.4 we will discuss what constraints on components must hold in order to give them meaning in the PF style. That is, we will explicitly define the domain of the function $\mathcal{M}^{PF}_{Comp}$.

Connectors are given meaning in PF by interpreting them as pipes. The concrete syntax for pipes specifies the type of data transmitted. Two pipes are considered equivalent if they have the same alphabets. Of course, in the context of a set of interacting filters, the pipes are distinguished by the data ports they connect.

$$\begin{array}{l} \mathcal{M}^{PF}_{Conn} : Connector \pfun \mathbb{P}\ Pipe \\ \hline \forall\, c : Connector; p_1, p_2 : Pipe \mid p_1 \in \mathcal{M}^{PF}_{Conn}(c) \\ \quad \bullet\ p_2 \in \mathcal{M}^{PF}_{Conn}(c) \Leftrightarrow p_1.alphabet = p_2.alphabet \end{array}$$

We can now define the meaning of configurations in the PF style. Components are interpreted as filters and connectors as pipes. The attachments are realized semantically by equating pipe sources with unique filter outputs and pipe sinks with unique filter inputs. To do this we select appropriate filter or pipe elements from the equivalence classes defined by the meaning functions $\mathcal{M}^{PF}_{Comp}$ and $\mathcal{M}^{PF}_{Conn}$. In the syntactic domain, we declare that *reader* and *writer* are distinct roles for connectors. The reader roles are mapped to sink data ports of the pipe and the writer roles are mapped to source data ports.

$$reader, writer : ROLE$$

$$reader \neq writer$$

$$\mathcal{M}^{PF}_{Conf} : Configuration \pfun InteractingFilterSet$$

$$\begin{array}{l}
\forall\, cfg : \mathrm{dom}\, \mathcal{M}^{PF}_{Conf} \bullet \\
\quad (\mathcal{M}^{PF}_{Conf}(cfg)).filters = \{n : COMPNAME; c : Component; f : Filter \mid \\
\qquad (n, c) \in cfg.components \\
\qquad \land\ f \in \mathcal{M}^{PF}_{Comp}(c) \\
\qquad \land\ f.outputs \cup f.inputs = DataPort(\!|\{n\} \times c.ports|\!) \\
\qquad \bullet\ f\} \\
\quad \land \\
\quad (\mathcal{M}^{PF}_{Conf}(cfg)).pipes = \{n : CONNNAME; c : Connector; p : Pipe \mid \\
\qquad (n, c) \in cfg.connectors \\
\qquad \land\ p \in \mathcal{M}^{PF}_{Conn}(c) \\
\qquad \land\ p.source = DataPort(cfg.attachment(n, writer)) \\
\qquad \land\ p.sink = DataPort(cfg.attachment(n, reader)) \\
\qquad \bullet\ p\}
\end{array}$$

4.4 Syntactic Constraints

The final part of defining a style is to make explicit the syntactic preconditions that must be satisfied in order to translate to the semantic domain. Since the meaning functions are partial, only a subset of all components, connectors and configurations are given a meaning in the PF style. This corresponds to the intuition that only some architectural descriptions represent valid pipe-filter systems. In particular, for components we demand that the computation associated with the component can be defined using the concrete language of *FilterDescription* and that the named component ports can be realized as data ports of some filter. We can express these syntactic constraints in Z by use of schema inclusion in which the original specification of type *Component* is included in the specification of syntactically legal PF components and then further constrained. (See Appendix A for a discussion of schema inclusion.)

LegalPFComponent

$$Component$$

$$description \in FilterDescriptions$$

By specifying this explicit syntactic constraint, we are actually asserting two things. First, only component descriptions that satisfy this constraint can be legally interpreted as a filter. This is equivalent to asserting that the domain of $\mathcal{M}^{PF}_{Comp}$ is *LegalPFComponent*.

$$\mathrm{dom}\, \mathcal{M}^{PF}_{Comp} = LegalPFComponent$$

Second, this assertion results in a proof obligation that we have not invalidated our definition of $\mathcal{M}^{PF}_{Comp}$. In other words, we must prove that given any legal PF component, we can apply $\mathcal{M}^{PF}_{Comp}$ to obtain a filter. We must show that

$$\forall\, c : LegalPFComponent \bullet \mathcal{M}^{PF}_{Comp}(c) \neq \varnothing$$

This amounts to demonstrating that

$$\begin{array}{l}
\forall\, c : LegalPFComponent; n : COMPNAME \bullet \\
\quad \exists\, f : Filter \bullet DataPort(\!|\{n\} \times c.ports|\!) = f.inputs \cup f.outputs
\end{array}$$

or, in essence, that the function *DataPort* is reasonably constructed. Therefore, the domain restriction to $\mathcal{M}^{PF}_{Comp}$ is valid.

Similarly, we constrain the definition of connectors to be those having a concrete description interpretable as a stream alphabet and having only two roles, *source* and *sink*.

LegalPFConnector

$$Connector$$

$$description \in PipeDescriptions$$
$$roles = \{reader, writer\}$$

Once again, we formally restrict the meaning function to cover legal values.

$$\mathrm{dom}\, \mathcal{M}^{PF}_{Conn} = LegalPFConnector$$

This also results in a proof obligation. Since $\mathcal{M}^{PF}_{Conn}$ as defined could be total, however, the proof is trivial.

As one might expect, the constraints we enforce on configurations are more complex. For the pipe and filter style defined above these are:

1. Each named component is a legal filter.
2. Each named connector is a legal pipe.
3. Every pipe reader is attached to a unique filter input with the same alphabet.
4. Every pipe writer is attached to a unique filter output with the same alphabet.

In the following schema, the first two predicates below the line express the first two constraints above. The third predicate below states that all pipe roles are attached to some named ports. The fourth predicate says that the attachment function is injective, that is, no two roles can be attached to the same port instances. The last two predicates express the alphabet constraint.

LegalPFConfiguration

$$Configuration$$

$$\forall c : \mathrm{ran}\, components \bullet c \in LegalPFComponent$$
$$\forall c : \mathrm{ran}\, connectors \bullet c \in LegalPFConnector$$

$$\mathrm{dom}\, attachment = \mathrm{dom}\, connectors \times \{reader, writer\}$$
$$attachment \in RoleInst \rightarrowtail\!\!\!\!\!\!\rightarrow PortInst$$

$$\forall n : CONNNAME; n' : COMPNAME; port : PORT \bullet$$
$$\quad attachment(n, writer) = (n', port) \Rightarrow$$
$$\quad\quad (\exists fil : \mathcal{M}^{PF}_{Comp}(components(n')); pipe : \mathcal{M}^{PF}_{Conn}(connectors(n)) \bullet$$
$$\quad\quad\quad DataPort(n', port) \in fil.outputs \wedge fil.alphabet(DataPort(n', port)) = pipe.alphabet)$$

$$\forall n : CONNNAME; n' : COMPNAME; port : PORT \bullet$$
$$\quad attachment(n, reader) = (n', port) \Rightarrow$$
$$\quad\quad (\exists fil : \mathcal{M}^{PF}_{Comp}(components(n')); pipe : \mathcal{M}^{PF}_{Conn}(connectors(n)) \bullet$$
$$\quad\quad\quad DataPort(n', port) \in fil.inputs \wedge fil.alphabet(DataPort(n', port)) = pipe.alphabet)$$

A straightforward argument shows that any syntactically legal configuration can be assigned a meaning by $\mathcal{M}^{PF}_{Conf}$, so we restrict its domain to *LegalPFConfig*.

$$\mathrm{dom}\, \mathcal{M}^{PF}_{Conf} = LegalPFConfig$$

This concludes our formal definition of the PF style. In Section 6 we investigate other syntactic constraints that can be used to define PF substyles and discuss some analysis that can be performed on the semantic domain of PF.

5 Event System Style

In this section, we show how the same method of definition for the PF style can be used to describe another common architectural style, the event system with implicit invocation (ES). Event systems are increasingly important as a flexible tool integration technique, since they allow the implicit invocation of tools when some other tool announces an event[16, 15, 32].

For the purposes of this paper we will treat each component in an event system as an object with a private, internal state and a collection of methods that can be invoked externally to alter the state. A component responds to an incoming method by transforming its internal state and announcing some events. Connection in the system consists of an association between announced events and the methods that should be invoked when those events are announced. Event announcement by one object in the system, therefore, results implicitly in the invocation of another object's method. Figure 8 gives an overview of the event system architectural style.

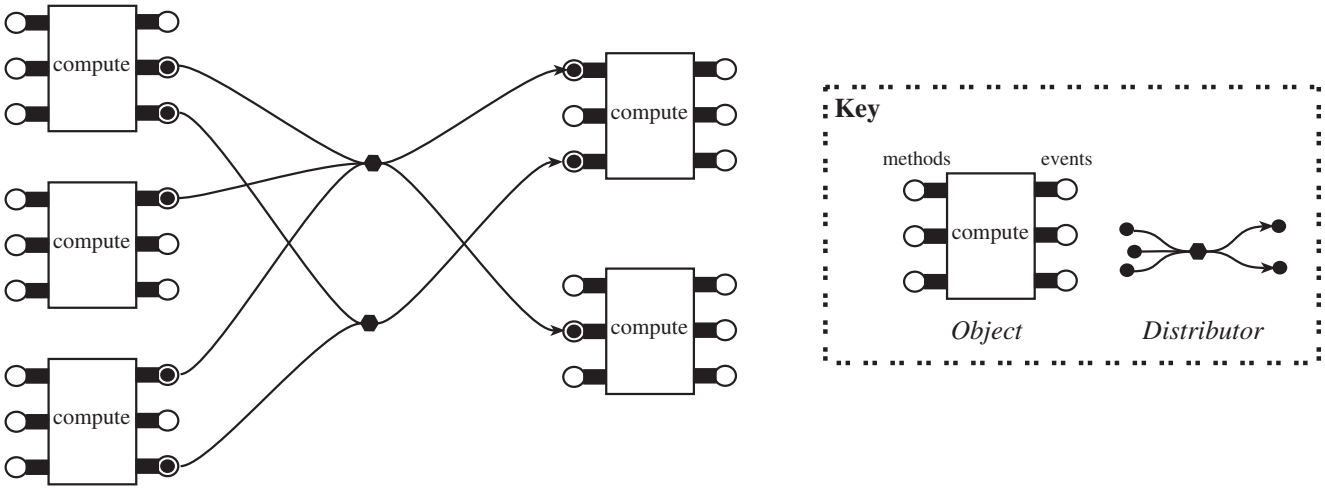

Figure 8: The event system style

5.1 Semantic Domain

The ES style interprets components as *objects* with a vocabulary of methods and events. Methods and events are the interaction points in the semantic model for event systems. Here we will model an object as a state machine with a transition function relating method invocations to state transitions and event announcement.

$$[METHOD, EVENT]$$

Object

$$
\begin{array}{l}
methods : \mathbb{P}\, METHOD \\
events : \mathbb{P}\, EVENT \\
states : \mathbb{P}\, STATE \\
start : STATE \\
transitions : (METHOD \times STATE) \pfun (STATE \times \mathbb{P}\, EVENT) \\
\hline
start \in states \\
\mathrm{dom}\, transitions = methods \times states \\
\mathrm{ran}\, transitions \subseteq \{\, s : states;\ es : \mathbb{P}\, EVENT \bullet (s, es) \,\}
\end{array}
$$

The ES style interprets connectors as *distributors*, which take announced events and transform them into method invocations. Our model of a distributor below is understood as saying that whenever any event in *events* is announced, then every method in *methods* must be invoked.

Distributor

$$
\begin{array}{l}
events : \mathbb{P}\, EVENT \\
methods : \mathbb{P}\, METHOD
\end{array}
$$

A collection of objects and distributors are joined to form a set of interacting objects. The overall binding of methods to events is derived from the bindings of the individual distributors in the system. There are two constraints we want to enforce. First, there can be no name clash between the local methods of the objects. Second, distributors can only bind events and methods that are defined in the system. This second semantic constraint means that we do not allow an event to be announced from some source outside the system and we do not allow method invocations on objects outside the system.

InteractingObjectSet

$$
\begin{array}{l}
objects : \mathbb{P}\, Object \\
distributors : \mathbb{P}\, Distributor \\
binding : EVENT \rightleftarrow METHOD \\
\hline
\forall\, o_1, o_2 : objects \mid o_1 \neq o_2 \bullet o_1.methods \cap o_2.methods = \emptyset \\
binding = \bigcup_{\{\, d : distributors \,\}} d.events \times d.methods \\
\forall\, e : \mathrm{dom}\, binding \bullet \exists\, o : objects \bullet e \in o.events \\
\forall\, m : \mathrm{ran}\, binding \bullet \exists\, o : objects \bullet e \in o.methods
\end{array}
$$

At any point in time, each object in the system will be in some legal state and the system will have some methods that have been invoked but not executed and some events that have been announced and not yet distributed. Since more than one occurrence of the same event or method can be pending, we model announced events and invoked methods as bags (see Appendix A).

IOState

$$
\begin{array}{l}
InteractingObjectSet \\
state : Object \pfun STATE \\
invoked : \mathrm{bag}\, METHOD \\
announced : \mathrm{bag}\, EVENT \\
\hline
\mathrm{dom}\, state = objects \\
\forall\, o : \mathrm{dom}\, state \bullet state(o) \in o.states
\end{array}
$$

A change in the system results when either a single object performs one of its pending invoked methods or when an announced event is distributed as method invocations to the relevant objects.

When an invoked method, that is, a method contained in the bag of invoked methods (bag membership indicated by $\sqsubseteq$), is performed by an object, the internal state of the object changes and a set of events are announced, as defined by the object's transition relation. The method is no longer pending (removal indicated with bag subtraction operator $\uplus$) and the announced events are suitably augmented (using the bag union operator $\uplus$).

IO_ObjectStep

$$
\begin{array}{l}
\Delta InteractingObjectStep \\
\Xi InteractingObjectSet \\
o? : Object \\
m? : METHOD \\
es! : \mathbb{P}\ EVENT \\
\hline
o? \in objects \\
m? \sqsubseteq invoked \\
m? \in o?.methods \\
((m?, state(o?)), (state'(o?), es!)) \in o?.transitions \\
(objects - \{\ o?\ \}) \lhd state = (objects - \{\ o?\ \}) \lhd state' \\
invoked' = invoked \uplus \{\ m? \mapsto 1\ \} \\
announced' = announced \uplus \{\ e : es! \bullet e \mapsto 1\ \}
\end{array}
$$

When an announced event is distributed, all methods bound to the event are pending.

IO_DistributorStep

$$
\begin{array}{l}
\Delta InteractingObjectStep \\
\Xi InteractingObjectSet \\
e? : EVENT \\
\hline
e? \sqsubseteq announced \\
e? \in events \\
state' = state \\
invoked' = invoked \uplus \{\ m : binding(\!|\{\ e?\ \}|\!) \bullet m \mapsto 1\ \} \\
announced = announced \uplus \{\ e? \mapsto 1\ \}
\end{array}
$$

A step in the behavior of a set of interacting objects is either a computation by one of its objects or it is a distribution of an announced event.

$$IO_Step \mathrel{\widehat=} IO_ObjectStep \lor IO_DistributorStep$$

5.2 Concrete Syntax

A concrete syntax for events systems can be developed as an extension of regular programming languages [37]. The details of these extensions are not particularly important for this discussion. These concrete descriptions define a subset of allowable computation and communication descriptions.

$$
\begin{array}{l}
ObjectDescriptions : \mathbb{P}\ COMPDESC \\
DistributorDescriptions : \mathbb{P}\ CONNDESC
\end{array}
$$

For example, Figure 9 illustrates a concrete syntax for the communication description extension that allows an Ada package interface to specify events announced by that package and the method to be invoked when an event is announced by some other package [17].

```
for Package_1
   declare Event_1 X : Integer;
   declare Event_2
   when Event_3 => Method_1 B
end for Package_1
for Package_2
   declare Event_3 A,B : Integer;
   when Event_1 => Method_2 X
   when Event_2 => Method_4
end for Package_2
```

Figure 9: Event System Description Example

5.3 Meaning Functions

The definition of meaning functions for ES proceeds exactly as for PF. The meaning function for ES components, written $\mathcal{M}^{ES}_{Comp}$, associates the syntactic elements of *Component* with equivalence classes of objects. Equivalence between objects is denoted by $\equiv_{obj}$.

To complete the mapping from syntax to semantics, we need to link ports and roles (the syntactic elements) to methods and events (the semantic interaction points). We want methods and events to be uniquely associated with object instances. Therefore, named port instances are identified as either a method or event, but not both.

$$
\begin{array}{|l}
EventasPort : PortInst \rightarrowtail\!\!\!

$$\mathcal{M}^{ES}_{Conn} : Connector \pfun \mathbb{P}\, Distributor$$

$$\forall\, c : \mathrm{dom}\, \mathcal{M}^{ES}_{Conn};\ d : Distributor \mid d \in \mathcal{M}^{ES}_{Conn}(c) \bullet$$
$$\#(d.events) = \#(c.roles \cap EventRoles) \land \#(d.methods) = \#(c.roles \cap MethodRoles)$$

The meaning of a configuration is derived from the meaning of its components, its connectors, and the attachment function. The attachment links events announced by an object to the same event received by one or more distributors. Also the attachment links methods received by an object to the same method invoked by one or more distributors.

$$\mathcal{M}^{ES}_{Conf} : Configuration \pfun InteractingObjectSet$$

$$\forall\, cfg : \mathrm{dom}\, \mathcal{M}^{ES}_{Conf} \bullet$$
$$(\mathcal{M}^{ES}_{Conf}(cfg)).objects =$$
$$\{n : \mathrm{dom}\, cfg.components;\ c : Component;\ o : Object \mid$$
$$cfg.components(n) = c$$
$$\land\ o \in \mathcal{M}^{ES}_{Comp}(c)$$
$$\land\ EventasPort^{\sim}(\!|o.events|\!) \cup MethodasPort^{\sim}(\!|o.methods|\!) = \{n\} \times c.ports$$
$$\bullet\ o\}$$
$$\land$$
$$(\mathcal{M}^{ES}_{Conf}(cfg)).distributors =$$
$$\{n : \mathrm{dom}\, cfg.connectors;\ c : Connector;\ d : Distributor \mid$$
$$cfg.connectors(n) = c$$
$$\land\ d \in \mathcal{M}^{ES}_{Conn}(c)$$
$$\land\ d.events = \{n' : COMPNAME;\ p : PORT$$
$$\mid \exists\, r : c.roles \cap EventRoles \bullet cfg.attachment(n, r) = (n', p)$$
$$\bullet\ EventasPort(n', p)\}$$
$$\land\ d.methods = \{n' : COMPNAME;\ p : PORT$$
$$\mid \exists\, r : c.roles \cap MethodRoles \bullet cfg.attachment(n, r) = (n', p)$$
$$\bullet\ MethodasPort(n', p)\}$$
$$\bullet\ d\}$$

5.4 Syntactic Constraints

The syntactic constraints in the ES style can be expressed by making explicit the domain for the meaning functions. For components, we simply restrict interpretation to those whose computation can be described using the concrete language of *ObjectDescriptions*.

LegalObject

$$Component$$

$$description \in ObjectDescriptions$$

Similarly for distributors, we restrict the abstract syntax to include only those connectors whose protocol can be described by the language of *DistributorDescriptions*.

LegalDistributor

$$Connector$$

$$description \in DistributorDescriptions$$

A legal configuration is one in which the components are legal objects, the connectors are legal distributors, and attachments only occur between event roles and event ports or between method roles and method

ports. Furthermore, since we do not allow dangling event-method bindings in the semantic model, we must ensure syntactically that there are no unattached roles in the configuration.

LegalESConfig

Configuration

$\forall\, c : \mathrm{ran}\, components \bullet c \in LegalObject$
$\forall\, c : \mathrm{ran}\, connectors \bullet c \in LegalDistributor$

$\forall\, n : CONNNAME;\ m : COMPNAME;\ role : ROLE;\ port : PORT \mid$
$\quad ((n, role), (m, port)) \in attachment \bullet$
$\quad\quad role \in EventRoles \Leftrightarrow (m, port) \in \mathrm{dom}\, EventasPort \wedge$
$\quad\quad role \in MethodRoles \Leftrightarrow (m, port) \in \mathrm{dom}\, MethodasPort$

$\forall\, cn : \mathrm{dom}\, connectors;\ r : connectors(cn).roles \bullet (cn, r) \in \mathrm{dom}\, attachment$

The domains of the meaning functions are accordingly defined.

$\mathrm{dom}\, \mathcal{M}^{ES}_{Comp} = LegalObject$
$\mathrm{dom}\, \mathcal{M}^{ES}_{Conn} = LegalDistributor$
$\mathrm{dom}\, \mathcal{M}^{ES}_{Conf} = LegalESConfig$

6 Analysis Using Architectural Style

One of the main reasons to formalize architectural style is to gain analytic leverage. In this section we present two examples of the kind of analysis of an architectural style that is possible within our formal framework.

6.1 Defining Architectural Substyles

It is common for one style to be understood in terms of another. Many of these *substyles* can be understood as additional constraints on the syntax of the more general style. For example, in the PF style we can identify the following common substyles:

- systems without feedback loops, or cycles;
- a pipeline; and
- only "fan-out" components.

The nature of pipes permits us to consider the topology of a PF configuration as a directed graph. We can derive the connection between two components by determining if any of their ports are attached to a common pipe.

PFGraph

LegalPFConfig
$connect : COMPNAME \leftrightarrow COMPNAME$

$connect =$
$\quad \{(c_1, p_1), (c_2, p_2) : PortInst;\ pipe : \mathrm{dom}\, connectors \mid$
$\quad\quad attachment(pipe, writer) = (c_1, p_1)$
$\quad\quad \wedge\ attachment(pipe, reader) = (c_2, p_2)$
$\quad\quad\quad \bullet (c_1, c_2)\}$

A PF system with no feedback loops is one in which the connection graph is acyclic.

Acyclic

$$PFGraph$$

$$\mathrm{id}\, COMPNAME \cap connect^{+} = \varnothing$$

To express acyclic pipe-filter architectures as an independent style, we restrict the meaning function $\mathcal{M}^{PF}_{Conf}$ to configurations satisfying *Acyclic*. The other meaning functions are the same as for the general PF style.

$$\mathcal{M}^{Acyclic}_{Comp} : Component \pfun \mathbb{P}\, Filter$$
$$\mathcal{M}^{Acyclic}_{Conn} : Connector \pfun \mathbb{P}\, Pipe$$
$$\mathcal{M}^{Acyclic}_{Conf} : Configuration \pfun InteractingFilterSet$$

$$\mathcal{M}^{Acyclic}_{Comp} = \mathcal{M}^{PF}_{Comp}$$
$$\mathcal{M}^{Acyclic}_{Conn} = \mathcal{M}^{PF}_{Conn}$$
$$\mathcal{M}^{Acyclic}_{Conf} = \{Acyclic \bullet \theta\, Configuration\} \lhd \mathcal{M}^{PF}_{Conf}$$

Restriction to a pipeline means that we can view the connection graph as a sequence of components, with each component in the pipeline sequence connected to the component after it in the pipeline.

Pipeline

$$PFGraph$$

$$\exists\, filters : \mathrm{seq}\, COMPNAME \mid \mathrm{ran}\, filters = \mathrm{dom}\, components \bullet$$
$$\quad connect = \{i : 1 \,..\, (\#filters - 1) \bullet (filters(i), filters(i+1))\}$$

A PF substyle allowing only fan-out has a connection graph whose inverse is a function, that is, components are connected to a unique parent component that provides its input.

FanOut

$$PFGraph$$

$$connect^{\sim} \in COMPNAME \pfun COMPNAME$$

Garlan and Notkin have used the event system model to investigate the differences between various implementations of an implicit invocation mechanism [16]. Their examples concentrate on restrictions to the kinds of events that objects can announce and the form of the event to method binding that a distributor allows. Since we have left the interpretation of events and methods open and allow distributors to bind events to methods arbitrarily, all of those styles are substyles of ES as it appears in this paper.

We can specify syntactic constraints that limit the topology of an event system the same way we did for PF. To show the generality of this kind of syntactic substyling, we will generalize the approach used for PF. We first define the connectivity for any syntactic configuration. This requires that we identify directionality in the roles. Some roles in the system will support outward flow of information from a component port and others will support an inward flow to a component port. The connection graph indicates when an instance of a component has one of its outbound ports connected to an inbound port of another component instance.

ArchGraph

$$
\begin{array}{l}
Configuration \\
connect : COMPNAME \leftrightarrow COMPNAME \\
outbound : \mathbb{P}\, ROLE \\
inbound : \mathbb{P}\, ROLE \\
\hline
connect = \\
\{ c_1, c_2 : \mathrm{dom}\, components; \; p_1, p_2 : PORT; \; r_{out} : outbound; \; r_{in} : inbound; \; n : \mathrm{dom}\, connectors \\
\quad \mid attachment(n, r_{out}) = (c_1, p_1) \land attachment(n, r_{in}) = (c_2, p_2) \\
\quad\quad \bullet \; (c_1, c_2) \}
\end{array}
$$

PFGraph can now be rewritten as a specialization of *ArchGraph* by indicating that the *writer* role is the only outbound role and the *reader* role is the only inbound role.

PFGraph

$$
\begin{array}{l}
ArchGraph \\
LegalPFConfig \\
\hline
outbound = \{ writer \} \\
inbound = \{ reader \}
\end{array}
$$

Similarly, for ES, we can define the connectivity graph by indicating that the event roles are the outbound roles and method roles are the inbound roles.

ESGraph

$$
\begin{array}{l}
ArchGraph \\
LegalESConfig \\
\hline
outbound = EventRoles \\
inbound = MethodRoles
\end{array}
$$

An architectural topology with no feedback is one in which the connection graph is acyclic.

AcyclicArch

$$
\begin{array}{l}
ArchGraph \\
\hline
\mathrm{id}\, COMPNAME \cap connect^{+} = \varnothing
\end{array}
$$

The acyclic event system is easily derived from this.

$$AcyclicES \mathrel{\widehat{=}} ESGraph \land AcyclicArch$$

We can also generalize the other topological constraints (pipeline and fan-out) in this way.

Another substyle of ES is one with a global event name space. In the current semantic model, events are uniquely associated to objects, and so they are treated independently with respect to distribution. In the global events substyle, we would like to treat events from different objects in the same way, meaning that if either event is announced, the same set of methods are invoked in the system. There are two ways we can go about defining this substyle. We can either adjust the semantic model and the meaning functions for ES, or we can add further constraints on legal ES configurations. We will demonstrate the latter.

In the global events substyle, we want instances of the same port to be treated the same way, that is, if one instance is attached to a connector, then the other instance is also attached to that same connector. Given what attachment means in ES in terms of event distribution, this constraint means that the event

from either component will result in the same distribution, or the events are essentially the same. This syntactic constraint is defined below.

GlobalEvents

LegalESConfig

$$
\begin{array}{l}
\forall\, n_1, n_2 : COMPNAME; p : PORT \mid \\
\quad (n_1, p) \in \mathrm{dom}\, EventasPort \\
\quad \wedge\ p \in (components(n_1)).ports \\
\quad \wedge\ p \in (components(n_2)).ports \\
\quad \bullet\ (\forall\, d : CONNNAME \bullet \\
\qquad (\exists\, r_1 : ROLE \bullet attachment(d, r_1) = (n_1, p)) \Leftrightarrow \\
\qquad\quad (\exists\, r_2 : ROLE \bullet attachment(d, r_2) = (n_2, p)))
\end{array}
$$

6.2 Relating Semantic Domains

One desirable property of an architectural description is hierarchy. In a hierarchical description components or connectors may themselves be represented as a configuration. For example, in the pipe and filter style, we might want to allow one filter to be expandable into a configuration of pipes and filters. By defining a style formally, we can understand what properties of the semantic domain might make this kind of description meaningful.

For example, Allen and Garlan showed formally that in the pipe and filter style it is semantically meaningful to decompose a component (filter) into a configuration of pipes and filters [3]. In their treatment, the decomposition is meaningful when the behavior of the unbound ports of the associated configuration matches the behavior of ports of an equivalent filter. In brief, the proof consists of the construction of a relation between a set of interacting filters and a single filter.

$$_\, collapse_{PF}\, _ : InteractingFilterSet \leftrightarrow Filter$$

This result means, for example, that we can now expand the concrete description language of filters to include hierarchical decomposition without altering our useful and intuitive understanding of the PF style.

This result leads us to ask whether a similar result holds for any other style. For example, in the event system style, does there exist a similar relation between collections of interacting objects and single objects? In other words, does there exist a relation

$$_\, collapse_{ES}\, _ : InteractingObjectSet \leftrightarrow Object$$

such that the external method/event behavior of the set of objects matches that of the collapsed object?

It turns out that there is not any such relation that covers all event systems. When a set of interacting objects is collapsed into a single object, any event-method connections internal to the set of objects will result in a computation that cannot be made to correspond to any visible method invocation. This is because the operational semantics as we have defined it makes method invocation *atomic*. That is, all of the effects of a method invocation are computed in one step, with no other computations intervening. Furthermore, no computation can occur except one that is the result of a method invocation. In an *InteractingObjectSet*, on the other hand, a method invocation can result in an event announcement that is then distributed to a second method invocation. When the object set is collapsed the distributor that triggers the second method invocation is hidden, so that the second method invocation appears to occur without any cause, which is not possible in any atomic object.

This result is useful because it tells us that if we want to provide hierarchical event systems we must do one of two things. Either we have to change the semantic model or we have to find ways to restrict the class of descriptions to a subset that allows hierarchical decomposition. In the former case we would need to view method invocation as non-atomic. In the latter case we might restrict decompositions to be configurations that do not have any internal event-method bindings.

7 Conclusion

We have argued that a formal approach to architectural style permits the precise interpretation and analysis of architectural descriptions. This has two important benefits. First, precision facilitates effective communication about systems at the architectural level. Misunderstandings inherent in ambiguous specifications can be avoided without abandoning the architectural paradigm. Second, a formal understanding of classes of systems permits the development of specialized analysis techniques as well as comparison between styles.

In addition to these immediate benefits, a precise understanding of style represents a necessary first step toward automated support for software architectural design and development. Through an understanding of both the structural constraints and the semantic underpinnings of architectures, tools and environments can be developed that effectively support the design process. As a first step in this direction, we have developed a software environment framework based on this model for style definition [13]. The common elements of component, connector, configuration, and hierarchy are directly supported by the environment, while its open structure supports the development and integration of tools that take advantage of style-specific structural and semantic properties. Because of the generality of structured style definition, tools developed for one style may be reused for any style that has certain properties in common with the original.

Acknowledgments

An shorter version of this paper appeared as "Using Style to Understand Descriptions of Software Architecture," in *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, Dec 1994. The paper also expands on material previously appearing in conference proceedings [16] and [2].

The authors would like to thank various colleagues whose comments on this work have helped us to clarify our thoughts, especially Dave Wile, Marc Graham, Mary Shaw, Jeannette Wing, Daniel Jackson, John Ockerbloom and Amy Moormann Zaremski.

The research reported here was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330; by National Science Foundation Grant CCR-9109469; and by a grant from Siemens Corporate Research. 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 US Department of Defense, the United States Government, the National Science Foundation, or Siemens Corporation. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

References

[1] Alexander, C., et al. *A Pattern Language*. Oxford University Press, New York, 1977.

[2] Allen, R., and Garlan, D. A formal approach to software architectures. In *Proceedings of IFIP'92* (September 1992), J. van Leeuwen, Ed., Elsevier Science Publishers B.V.

[3] Allen, R., and Garlan, D. Towards formalized software architectures. Tech. Rep. CMU-CS-92-163, Carnegie Mellon University, School of Computer Science, July 1992.

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

[5] Allen, R., and Garlan, D. Formalizing architectural connection. In *Proceedings of the Sixteenth International Conference on Software Engineering* (Sorrento, Italy, May 1994), pp. 71–80.

[6] Berry, G., and Boudol, G. The chemical abstract machine. *Theoretical Computer Science*, 96 (1992), 217–248.

[7] *Proceedings of the Workshop on Domain-Specific Software Architectures* (Hidden Valley, PA, July 1990), Software Engineering Institute.

[8] Earl, A. A reference model for computer assisted software engineering environment frameworks. Tech. Rep. HPL-SEG-TN-90-11, Hewlett Packard Laboratories, Bristol, England, August 1990.

[9] Freeman, P., and Wasserman, A. Tutorial on software design techniques, 1976.

[10] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design patterns: Abstractions and reuse of object-oriented design. In *Proceedings of ECOOP* (July 1993), Springer Verlag LNCS 707.

[11] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Micro-Architectures for Reusable Object-Oriented Design.* Addison-Wesley, 1994.

[12] Garlan, D., Allen, R., and Ockerbloom, J. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering* (December 1994), ACM Press.

[13] Garlan, D., Allen, R., and Ockerbloom, J. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering* (December 1994), ACM Press.

[14] Garlan, D., and Delisle, N. Formal specifications as reusable frameworks. In *VDM'90: VDM and Z – Formal Methods in Software Development* (Kiel, Germany, April 1990), Springer-Verlag, LNCS 428, pp. 150–163.

[15] Garlan, D., Kaiser, G. E., and Notkin, D. Using tool abstraction to compose systems. *IEEE Computer 25*, 6 (June 1992).

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

[17] Garlan, D., and Scott, C. Adding implicit invocation to traditional programming languages. In *Proceedings of the Fifteenth International Conference on Software Engineering* (Baltimore, MD, May 1993).

[18] Garlan, D., and Shaw, M. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering, Volume I* (New Jersey, 1993), V. Ambriola and G. Tortora, Eds., World Scientific Publishing Company.

[19] Hayes-Roth, B., Pfleger, K., Lalanda, P., Morignot, P., and Balabanovic, M. A domain-specific software architecture for adaptive intelligent systems. *IEEE Transactions on Software Engineering* (1995). To appear.

[20] Hayes-Roth, F. Rule-based systems. *Communications of the ACM 28*, 9 (September 1985), 921–932.

[21] Hoare, C. *Communicating Sequential Processes.* Prentice Hall, 1985.

[22] Inverardi, P., and Wolf, A. Formal specification and analysis of software architectures using the chemical, abstract machine model. *IEEE Transactions on Software Engineering, to appear* (1995).

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

[24] Luckham, D. C., Augustin, L. M., Kenney, J. J., Veera, J., Bryan, D., and Mann, W. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering, to appear* (1995).

[25] Mettala, E., and Graham, M. H. The domain-specific software architecture program. Tech. Rep. CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, June 1992.

[26] Milner, R., Tofte, M., and Harper., R. *The definition of Standard ML*. MIT Press, 1990.

[27] Moriconi, M., and Qian, X. Correctness and composition of software architectures. In *Proceedings of ACM SIGSOFT'94: Symposium on Foundations of Software Engineering* (New Orleans, Louisiana, December 1994).

[28] Moriconi, M., Qian, X., and Riemenschneider, R. Correct architecture refinement. *IEEE Transactions on Software Engineering* (1995). To appear.

[29] Morris, C. R., and Ferguson, C. H. How architecture wins technology wars. *Harvard Business Review 71*, 2 (March-April 1993).

[30] Perry, D. E., and Wolf, A. L. Foundations for the study of software architecture. *Software Engineering Notes 17*, 4 (1992), 40–52.

[31] Pree, W. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, ACM Press, 1995.

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

[33] Shaw, M. Larger scale systems require higher level abstractions. *Proceedings Fifth International Workshop on Software Specification and Design, IEEE Computer Society, Software Engineering Notes 14*, 3 (May 1989), 143–146.

[34] Shaw, M., et al. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering* (April 1995). To appear.

[35] Shaw, M., and Garlan, D. Characteristics of higher-level languages for software architecture, 1994.

[36] Spivey, J. *The Z Notation: A Reference Manual*. Prentice Hall, 1989.

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

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

A A Guide to the Z Notation Used in this Paper

The Z notation is a mathematical language developed mainly at the Programming Research Group at the University of Oxford over the last 15 years. The mathematical roots of Z are in first order logic and set theory. The notation uses standard logical connectives ($\wedge$, $\vee$, $\Rightarrow$, *etc.*) and set-theoretic operations ($\in$, $\cup$, $\cap$, *etc.*) with their standard semantics. Using the language of Z, we can provide a model of a mathematical object. That these objects bear a resemblance to computational objects reflects the intention that Z be used as a specification language for software engineering. In this appendix, we describe the basics of the Z notation used in this paper. The standard reference for practitioners of Z, and the basis for our use of Z, is Spivey's reference manual [36].

A Z specification consists of sections of mathematical text interspersed with prose. The mathematical text is a collection of types together with some predicates that must hold on the values of each type. Types in Z are sets of values. Z provides some fundamental types in its basic toolkit that are primitive, such as $\mathbb{N}$ for natural numbers and $\mathbb{Z}$ for integers. In addition, we can introduce further primitive types, called given

types, by writing them in square brackets. By convention, given types are written in all capital letters. The construction of elements in a given type is not provided in a specification, usually because that level of detail is not necessary for the purposes of the specification. Prose surrounding the declaration of a given type should indicate the reason the specifier has introduced the type rather than use an existing type. For example, we could introduce two given sets to represent all possible authors and papers that those authors might write. For use in this appendix, no further information about authors or papers need me made explicit, so we write:

$$[AUTHOR, PAPER]$$

An element of a type is declared using a colon (:). So we would write $author : AUTHOR$ and read this as "*author* is of type *AUTHOR*", meaning *author* is an element in the set of values defined by *AUTHOR*. Since *AUTHOR* is a set, we could also write $author \in AUTHOR$, using the set membership function $\in$. Z uses the : notation when a variable is declared and $\in$ to express predicates over bound variables.

New types can also be defined by constructing them from primitive types using the following type constructors:

- $\mathbb{P}\, X$ is the set of all subsets with elements from type X, also called the powerset of X.
- $X \times Y$ is the type consisting of all ordered pairs (x, y) whose first element is of type X and whose second element is of type Y, also called the cross-product of X and Y.
- $\mathrm{seq}\, X$ is the set of all sequences, or lists, of elements from X, including empty and infinite sequences.
- $\mathrm{bag}\, X$ is the set of all bags of elements from X. A bag is a collection of elements from some base type in which the number of times an element occurs is significant.
- Relations and functions between types identify special subsets of the cross product type. The ones used in this paper are:
  - $X \leftrightarrow Y$ is the set of all relations between domain type X and range type Y. A relation is simply a subset of $X \times Y$.
  - $X \pfun Y$ is the set of all partial functions between X and Y. A partial function does not have to be defined on all elements of its domain type.
  - $X \fun Y$ is the set of all total functions. Total functions are defined on all elements of the domain type.
  - $X \pinj Y$ is the set of all partial functions from X to Y whose inverse is a partial function from Y to X (also called 1-1 or injective).
  - $X \inj Y$ denotes the total injective functions from X to Y.
  - $X \bij Y$ denotes the bijective functions from X to Y, i.e., the functions from X to Y that are a 1-1 correspondence (total, injective and surjective).

Part of the power of Z types, which often confuses those unfamiliar with the notation, is that many of the constructed types are derived from each other. Functions and relations are derived from the cross-product constructor. Sequences and bags, in turn, are derived from partial functions. For instance, the type $\mathrm{seq}\, X$ is a subsect of the finite partial functions from the natural numbers ($\mathbb{N}$) to the type X, with the constraint that the domain of the function be a segment $1 \mathinner{..} n$ of natural numbers, for some n. The type $\mathrm{bag}\, X$ indicates a partial function from the type X to the positive natural numbers ($\mathbb{N}_1$, not including 0), reflecting the count of elements in X that are in the bag. Because these types are derived from more primitive types, it is possible to manipulate them using operations defined on the more primitive type. For example, since a bag is a function, we can ask about its domain, or use functional overriding to change the contents of a particular bag.

Z has a special type constructor, called the *schema*, an abstract version of the Pascal record or the C struct type constructors. A schema defines a binding of identifiers (or variables) to their values in some type. For example, we could specify the type *Proceedings* as a schema for a typical conference proceedings. The information we might want to specify about a proceedings would be the set of all authors and an index from authors to the papers they wrote. We represent this binding in the boxed schema notation below.

Proceedings

$$authors : \mathbb{P}\, AUTHOR$$
$$index : AUTHOR \leftrightarrow PAPER$$

A "dot" notation is used to select elements of a schema type. So we could refer to the authors in the proceedings $sigsoft93 : Proceedings$ by writing $sigsoft93.authors$.

In addition to declaring the bindings between identifiers and values, a schema can specify invariants that must hold between the values of identifiers. In the boxed notation, these invariants are written under a dividing line. All common identifiers below the line are scoped by the declarations above the line. If we wanted to model the invariant that the set of authors in type *Proceedings* can and must include only those authors appearing in the index, we could state that *authors* is the domain of the *index* relation. We would write this as follows.

EssentialProceedings

$$authors : \mathbb{P}\, AUTHOR$$
$$index : AUTHOR \leftrightarrow PAPER$$

$$authors = \mathrm{dom}\, index$$

Z allows for schema inclusion to facilitate a more modular approach to a specification. In the above example, we could have introduced the invariant on the set of authors as

EssentialProceedings

$$Proceedings$$

$$authors = \mathrm{dom}\, index$$

including the declarations and invariants of *Proceedings* in the new schema *EssentialProceedings*. Z defines a calculus of schema operations of which inclusion is just one example. We do not use many schema operations in this paper, so we direct the interested reader to Spivey's reference manual.

In addition to the schema calculus for defining schema expressions, Z usage relies on some notational conventions for describing the behavior of state machines. The schema represents a binding from identifiers to values. We can view this binding as the static description of some state machine, that is, the view of the state machine at some point in time. Operations on the state machine are transitions from one legal state to another and can be described as a relationship between the values of identifiers before and after the operation. One of the most common conventions is the Δ convention for describing operations. If *Schema* is a schema type, then $\Delta Schema$ is notationally equivalent to two "copies" of *Schema*, one of which has all of its identifiers decorated with dashes ($'$) to indicate the state after the operation. So, we could write

ProceedingsOp

$$\Delta Proceedings$$

which is equivalent to

ProceedingsOp

$$Proceedings$$
$$Proceedings'$$

or

ProceedingsOp

$authors : \mathbb{P}\, AUTHOR$
$index : AUTHOR \leftrightarrow PAPER$
$authors' : \mathbb{P}\, AUTHOR$
$index' : AUTHOR \leftrightarrow PAPER$

Some other operations and notational conventions used in Z are:

- $Point == \mathbb{N} \times \mathbb{N}$ introduces the type $Point$ as a type synonym for the cross product. Type synonyms are a notational convenience.
- If f is a relation, function or sequence, then $\mathrm{dom}\, f$ is the domain of f and $\mathrm{ran}\, f$ is the range of f.
- If S is a set (or sequence), then $\#$ S is the size (or length) of S.
- $a \frown b$ is the concatenation of sequences a and b.
- If R is a relation, then $R^{\sim}$ is its relational inverse and R^{+} is its transitive closure. If S is a set of elements in the domain type of R, then $R(\!| S |\!)$ is the image over R of the set of elements in S, that is, the set of elements in the range type of R that are related to elements in S under R.
- If f and g are functions of the type $X \pfun Y$, then $f \oplus g$ is another function of type $X \pfun Y$ which agrees with g everywhere in X that g is defined. On the rest of its domain, it agrees with f.
- A function is understood as a mapping from one set to another. The expression $x \mapsto y$, indicates a mapping from an element in one set $(x : X)$ to an element in another $y : Y$. This 'maplet' notation is convenient when used in conjunction with functional overriding. The expression $f' = f \oplus \{ x \mapsto y \}$ indicates that the new function f' agrees with the old function f at every point in its domain except x, which is to be mapped to element y.
- $\forall\, decl \mid pred_1 \bullet pred_2$ is read "for all variables in $decl$ satisfying $pred_1$, we have that $pred_2$ holds."
- $\exists\, decl \mid pred_1 \bullet pred_2$ is read "there exist(s) variable(s) in $decl$ satisfying $pred_1$ such that $pred_2$ holds."
- $\{ decl \mid pred \bullet expression \}$ is a set comprehension for the set of values $expression$ ranging over variables in $decl$ satisfying the predicate $pred$.