Formalizing style to understand descriptions of software architecture

Gregory Abowd, Robert Allen and David Garlan
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

July 13, 1994

Abstract

The software architecture of most systems is described informally and diagrammatically by means of boxes and lines. In order for these descriptions to be meaningful at all, the diagrams are understood by interpreting the boxes and lines in specific, conventionalized ways. The imprecision 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.

Keywords: Software architecture, models of architectural style, formal specification, the Z notation

1 Introduction

Software architecture is an important level of description for software systems [15, 12]. The software architecture is an abstract model of a system, and so it is relevant to ask what kind of information resides in such a model. In practice, when designers discuss or present a software architecture, they are considering the system as a collection of interacting *components*. Components perform the primary computations of the system. Interactions, or *connections*, between components are high level communication abstractions. A variety of component and connector types are used to represent different forms of computation or interaction [11]. Examples of different component types are modules, filters, objects, and services. Examples of different connector types are pipes, procedure calls, message passing, and event broadcast. Our primary concern in this paper is understanding and improving upon the ways in which these varities of components and connectors are combined to provide meaningful descriptions of systems at the architectural level.

The majority of architectural descriptions are given informally and diagrammatically using boxes to represent components and lines to represent the connections. In order for these descriptions to be meaningful at all, there are several questions about the system which they must answer:

- What computation occurs in the boxes?
- Are the boxes somehow similar in behavior?
- what control/data relationships are manifest by the lines?
- What is the overall behavior of the system?
- Does the diagram make sense, that is, does it represent a legal configuration of boxes and lines?

Box and line diagrams in isolation do not provide all of the answers to these questions, so designers typically resort to conventional interpretations of the diagrams in order to provide those answers [7]. For example, for one system boxes might represent filters and lines might represent pipes connecting ports of those filters. In another, 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 usually distinguished by different graphical representations.

While useful in documenting system designs, such diagrams — even with their conventional interpretations — have a number of limitations. Their imprecision makes it difficult to attach unambiguous meanings to the descriptions. It is difficult to know when an implementation agrees with the more abstract description and, therefore, difficult to know how changes to one affects the other. Whereas the software architectural descriptions do provide abstraction away from unnecessary implementation detail, their lack of precision precludes the benefit of formal analysis. It is virtually impossible to reason formally about a single description, or to reasonably compare two different descriptions.

The most common solution to this problem of informal interpretation is to constrain the architectural notation so that it maps directly into a well-defined execution model . For example, interfaces to components can be described solely in terms of their procedure signatures, and connectors can be restricted to procedure call. Other execution models include tasks with interprocess communication and event-based systems [13]. When so constrained, descriptions can be mapped directly to facilities of a programming language or other executable implementations, and can thereby be given precise meanings.

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 by the target implementation language or system. If, for instance, architectural connections have to be phrased in terms of procedure calls, then higher-level interactions (such as protocols of communication) cannot be expressed directly. In addition, the relatively low level of description may make it difficult to reason about the architectural design.

A more acceptable approach is to accept the variety of conventional interpretations assigned to architectural diagrams and to create a framework for understanding and defining them more precisely. We argue that what is needed instead is a way to give conventionalized interpretations of architectural descriptions a more flexible and formal basis. Designers can use the abstractions that are appropriate to the architectural description at hand, but still have the precision of a formal model. We view the collection of conventions that are used to interpret a class of descriptions as defining an *architectural style*. A complete understanding of the meaning of an architectural description requires both a diagram (which details topological information) as well as the indication of style under which the description is to be understood. So, in Figure 1, we can

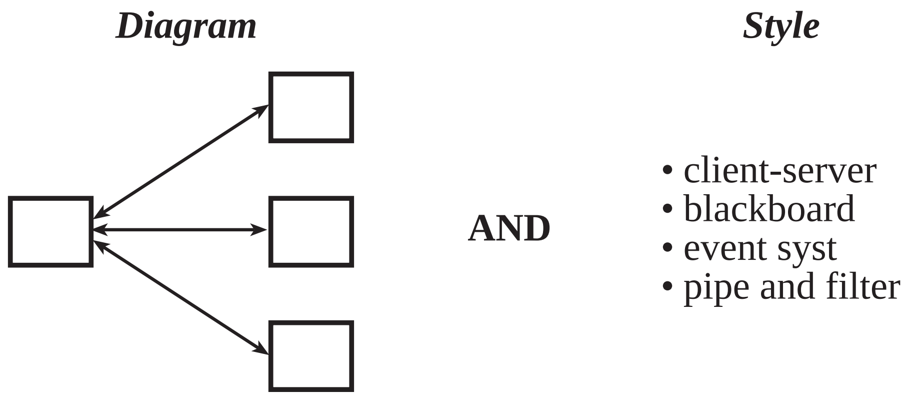

Figure 1: How style distinguishes similar descriptions

see that a diagram identifies the number and connectivity of computational entities, but it is the style which tells us what kinds of components should exist, the control/data relationships between components and other important information. For example, interpreting the diagram of Figure 1 under the client-server architectural style indicates to us two kinds of components with a request-reply protocol initiated by the clients that connects them. Furthermore, interpreting the same diagram under the pipe-filter style should indicate to us that the diagram is illegal, as pipes are not used for two-way connection between the components in that style.

In this paper, we will show that 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. The overview of our approach is depicted in Figure 2. The approach thus provides a framework in

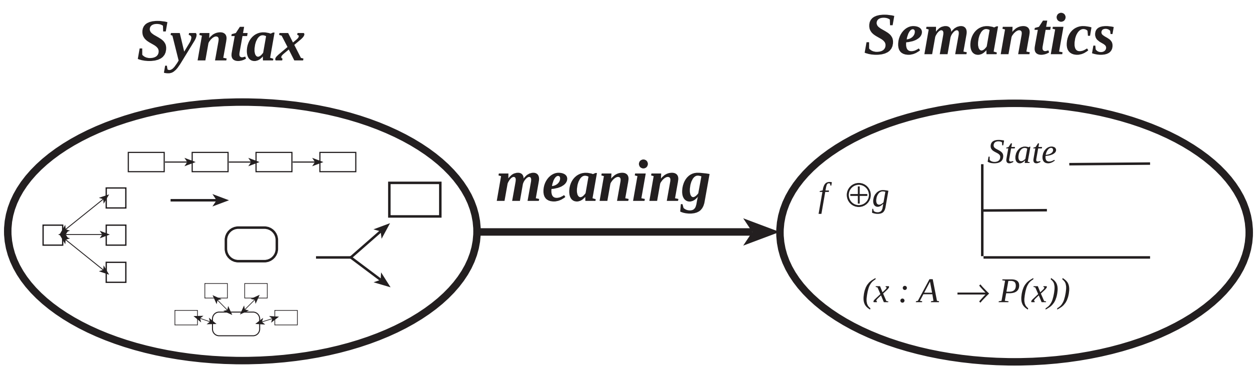

Figure 2: Approach to formalizing architectural style

which new styles can be defined by a similar set of definitions. The formal model further makes it possible to gain insight into the properties of a style and its relationships to other styles by direct analysis.

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 can be given meanings.

This work 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 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 use of conventionalized interpretations of software architectural descriptions is an extremely common practice ,which makes it a very important practice to understand and codify. It is only recently that this practice has been identified with architectural style. Garlan and Shaw [11] demonstrated the use of idioms or common patterns of architectural descriptions and how they can be used to provide varied solutions to a common design problem (such as Parnas' Key Word in Context problem [5]). The first mention of architectural style was by Perry and Wolf [15]. They introduced style as a means of capturing the similarities between instances of architectures. Our intuition behind style is very similar to both of these, but we wish to move beyond an intuitive or informal understanding of style toward a framework which suggests a uniform description of all styles and provides a rigorous means of comparing styles.

There are a growing number of industrial research and development efforts that are creating domain-specific architectural styles — or "reference architectures" — for specific product families [4, 6, 14]. To the extent that they formalize their architectural frameworks at all, the semantic descriptions produced by these efforts are typically developed from scratch, and each uses different, idiosyncratic conventions and semantic bases. Semantic descriptions are therefore difficult to develop and, having developed them, few comparisons can be made between different development efforts.

Other formal modeling of classes of architectural descriptions have been presented by the authors and other colleagues, in the form of a formalization for a class of signal processing systems [8], the pipe-filter style [1] and the implicit invocation style [9]. This previous work was a strong motivation for providing a unified framework for defining an architectural styles. In fact, the two examples of architectural styles are the pipe-filter style and the implicit invocation style (which we refer to as the event system style), to show that our framework is consistent with that previous work but also enables a much more structured approach to style definition.

Rice and Seidman have recently presented a formal model of module interconnection languages (or MILs) that can be used to understand and compare their differences [17].

1.2 Overview

In Section 2, we begin by outlining the method we will use to define an architectural style as a mapping from syntactic descriptions to a (style-specific) semantic model. In Section 3, we formalize the syntactic domain as an abstraction of the box and line diagrams that are prevalent in current informal architectural descriptions. 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 the architectural syntax can be mapped into that model by 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 this paper, we we use the Z specification language to express the formal model [18]. Appendix A summaries the Z notation that we use in this paper. 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. The use of any single notation for our work, therefore, is not of primary concern. We chose Z because it was simple to define the framework abstractly. It is quite possible that more appropriate formalisms could be chosen for parts of our examples. For instance, Allen and Garlan have recently explored the utility of process algebras for describing the protocols associated with conector types [3].

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 are the locus of communication or interactin between components; 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 strong 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 adopt the notion of architectural style as 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 component, connector and configuration. 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 the syntactic descriptions to their semantic counterparts. For style X, we would declare the meaning functions as partial functions from the abstract syntax to the semantic models.

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

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 $\nrightarrow$) to indicate that not every syntactic element in *Component* can be legally assigned a meaning in a given style. In fact, it is part of the definition of a style to determine those syntactic elements which can legally be assigned a meaning. This is done by defining explicitly the domain of the meaning functions (written as $\mathrm{dom}(\mathcal{M}^X_{Comp})$ in our generic example for components. Similar conventions are used for defining 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 carries a proof obligation to show that the meaning function is well-defined for all syntactic elements which meet the constraints. By making the constraints explicit we are precise about the descriptions that are reasonable in the style.

After we have formally defined an architectural style using the method outlined above, we have 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, which we exemplify by comparing different semantic models to see if they share similar properties.

To summarize, the steps we will follow are:

- formalize abstract syntax for architectures
- 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
- demonstrate analysis within and between formally defined architectural styles.

3 The Abstract Syntax of Software Architectures

From an abstract, generic point of view the basic syntactic elements of an architectural description are *components*, *connectors*, and *configurations* of components and connectors.

3.1 Components

Components are the active, computational entities of an architecture (see Figure 3). They accomplish tasks

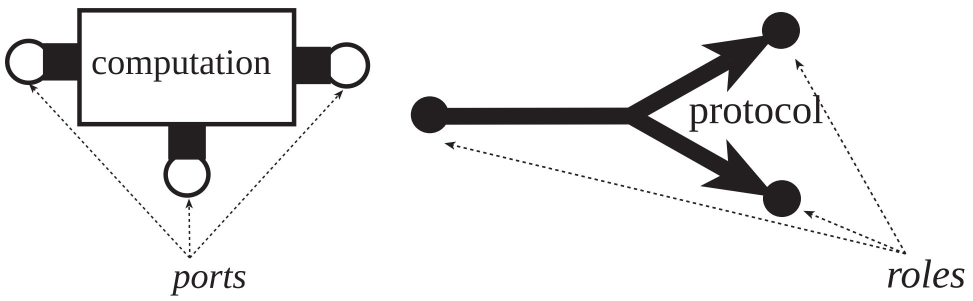

Figure 3: A component and a connector

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*. We can also differentiate between components with the same port interface based on a description of the computation they perform. At the abstract level of a component, we model this reference to computational behavior with a placeholder for some computational description.

For the moment, we are not concerned with details of the construction of ports or the computational description for components, so we model these as given sets. An architectural component, as a syntactic entity, is modeled as a collection of ports together with a description of its computation. We use the Z schema construct to define the type *Component* to represent this syntactic element of an architectural description.

$[PORT, COMPDESC]$

Component
$ports : \mathbb{P}\ PORT$
$description : COMPDESC$

3.2 Connectors

Connectors define the communication between components (see Figure 3. Each connector provides a way for a collection of ports to come into contact. A connector, rather than being bound unchangeably to specific ports on specific components, provides placeholders for these ports, as *roles* in the communication. The description of the precise communications protocol provided within a connector is separated from its interface, in the same way that the computation description in a component is separated from its port interface.

Again, we are not yet concerned with the details of roles or communication description, so we introduce them as given sets in this specification. An architectural connector is modeled as a collection of roles together with a description of its communication 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 4). Instances of components and connectors are provided by naming elements from the syntactic class.

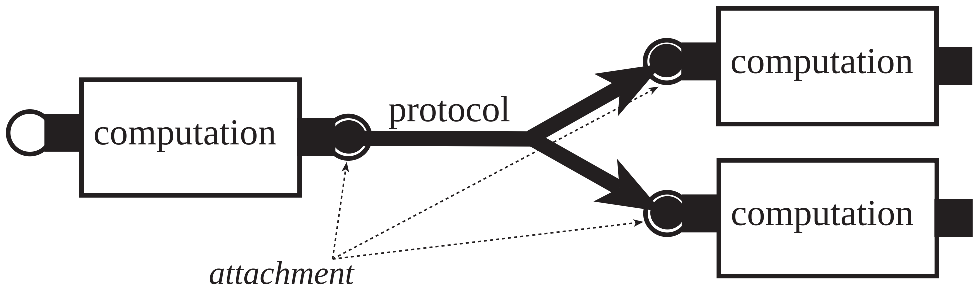

Figure 4: A configuration

We introduce two sets in order to name instances of components and connectors. These naming sets are also used to name instances of ports or roles associated with a component or connector, and so we introduce two type synonyms for convenience.

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

The interaction 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 communication protocol which are to be filled by various component ports. This intuition limits the kind of attachments allowed. 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 configuration is given below. Instances of components and connectors are modeled by a partial function from the naming set to the syntactic class. Attachments are modelled as a partial function from the roles of the connector instances to the ports of the component instances.

Configuration

$components : COMPNAME \rightarrowtail\!\!\!\!\!+ \; Component$
$connectors : CONNNAME \rightarrowtail\!\!\!\!\!+ \; Connector$
$attachment : RoleInst \rightarrowtail\!\!\!\!\!+ \; PortInst$

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

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

The schema *Configuration* includes two additional constraints (below the separating line) that must be satisfied by all configurations. The first constraint is a predicate that ensures that any role instance in the *attachment* is a role for some named connector in the configuration. The second constraint ensures a similar fact for port instances and the named components. Together, these two constraints enforce a lexical scoping on attachments within a configuration.

4 The Pipe-Filter Style

In this section, we show how the framework can be used to define the pipe-filter Style (*PF*). This style is representative of coarse-grained dataflow systems, such as those supported by Unix pipes. Figure 5 gives an overview of the pipe-filter architectural style.

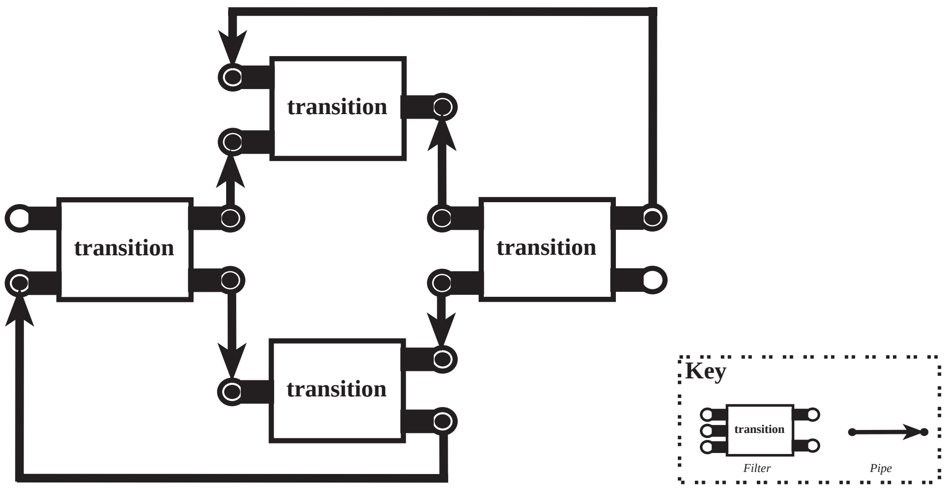

Figure 5: The pipe-filter style

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. In general, 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 [1, 2]. 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. These are to be distinguished from the ports that form the interface for unnamed 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

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

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

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

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 state, a change in the current state and the addition 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).

$$
\begin{array}{|l}
\hline
\textit{FilterStep} \\
\Delta \textit{FilterState} \\
\hline
f' = f \\
\exists\, in, out : DATAPORT \rightarrowtail\!\!\!\!\!\mapsto \operatorname{seq} DATA \\
\quad \bullet\ ((curstate, in), (curstate', out)) \in f.transitions \\
\qquad \land \forall\, p : f.inputs \\
\qquad\quad \bullet\ instate(p) = indata(p) \frown instate'(p) \\
\qquad \land \forall\, p : f.outputs \\
\qquad\quad \bullet\ outstate'(p) = outstate(p) \frown outdata(p) \\
\hline
\end{array}
$$

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.

$$
\begin{array}{|l}
\hline
\textit{Pipe} \\
source, sink : DATAPORT \\
alphabet : \mathbb{P}\, DATA \\
\hline
source \neq sink \\
\hline
\end{array}
$$

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.

$$
\begin{array}{|l}
\hline
\textit{PipeState} \\
p : Pipe \\
sourcedata : \operatorname{seq} DATA \\
sinkdata : \operatorname{seq} DATA \\
\hline
\operatorname{ran} sourcedata \subseteq p.alphabet \\
\operatorname{ran} sinkdata \subseteq p.alphabet \\
\hline
\end{array}
$$

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.

$$
\begin{array}{|l}
\hline
\textit{PipeStep} \\
\Delta \textit{PipeState} \\
\hline
p = p' \\
\exists\, deliver : \operatorname{seq} DATA \\
\quad \mid \#deliver > 0 \\
\quad \bullet\ \quad deliver \frown sourcedata' = sourcedata \\
\qquad \land\ \ sinkdata' = sinkdata \frown deliver \\
\hline
\end{array}
$$

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 6: Concrete Description of a Capitalizing Filter

InteractingFilterSet

$filters : \mathbb{P}\ Filter$
$pipes : \mathbb{P}\ Pipe$

$$\begin{aligned}
&\forall f_1, f_2 : filters \\
&\qquad \mid f_1 \neq f_2 \\
&\qquad \bullet (f_1.inputs \cup f_1.outputs) \cap (f_2.inputs \cup f_2.outputs) = \emptyset \\
&\forall p_1, p_2 : pipes \\
&\qquad \mid p_1 \neq p_2 \\
&\qquad \bullet \{p_1.source, p_1.sink\} \cap \{p_2.source, p_2.sink\} = \emptyset \\
&\forall p : pipes \\
&\qquad \bullet \exists f_1, f_2 : filters \\
&\qquad\qquad \bullet \quad p.source \in f_1.outputs \\
&\qquad\qquad\quad \wedge \; p.sink \in f_2.inputs \\
&\qquad\qquad\quad \wedge \; f_1.alphabet(p.source) = p.alphabet \\
&\qquad\qquad\quad \wedge \; f_2.alphabet(p.sink) = p.alphabet
\end{aligned}$$

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 in [2].

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.

$FilterDescriptions : \mathbb{P}\ COMPDESC$
$PipeDescriptions : \mathbb{P}\ CONNDESC$

For concreteness, Figure 6 illustrates the definition of a filter that capitalizes its character input stream using one notation developed for this style [2].

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:

$$\mathcal{M}^{X}_{Comp} : Component \rightarrow\!\!\!\!\!+ \; Comp^{X}_{sem}$$

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 $\mathbb{P}\ Filter$. In order 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.

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

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 dataports they connect.

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

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. Informally, the reader roles are mapped to sink data ports of the pipe and the writer roles are mapped to source data ports.

$$
\begin{array}{|l}
reader, writer : ROLE \\
\hline
reader \neq writer
\end{array}
$$

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

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 further details on schema inclusion.)

$$
\begin{array}{|l}
\text{--- } LegalPFComponent \text{ ---} \\
Component \\
\hline
description \in FilterDescriptions \\
\hline
\end{array}
$$

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*.

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

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

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

This amounts to demonstrating that

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

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 ∈ PipeDescriptions
│ roles = {reader, writer}
└────────────────────────────────────────────
```

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

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

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 output with the same alphabet.
4. Every pipe writer is attached to a unique filter input 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 sources and sinks are attached to some named ports. The fourth predicate says that the attachment function is injective, that is, no two sources or sinks can be attached to the same port instances. The last two predicates express the alphabet constraint.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{LegalPFConfiguration}} \\
\textit{Configuration} \\
\hline
\forall\, c : \operatorname{ran} \textit{components} \bullet c \in \textit{LegalPFComponent} \\
\forall\, c : \operatorname{ran} \textit{connectors} \bullet c \in \textit{LegalPFConnector} \\
\\
\operatorname{dom} \textit{attachment} = \operatorname{dom} \textit{connectors} \times \{\textit{reader}, \textit{writer}\} \\
\textit{attachment} \in \textit{RoleInst} \mathrel{\rightarrowtail\!\!\!\!\!\!+\;\;} \textit{PortInst} \\
\\
\forall\, n : \textit{CONNNAME};\ n' : \textit{COMPNAME};\ \textit{port} : \textit{PORT} \\
\bullet\ \textit{attachment}(n, \textit{writer}) = (n', \textit{port}) \Rightarrow \\
\qquad (\exists\, \textit{fil} : \mathcal{M}^{PF}_{Comp}(\textit{components}(n')); \\
\qquad\qquad \textit{pipe} : \mathcal{M}^{PF}_{Conn}(\textit{connectors}(n)) \\
\qquad\qquad \bullet\quad \textit{DataPort}(n', \textit{port}) \in \textit{fil.outputs} \\
\qquad\qquad\quad \wedge \textit{fil.alphabet}(\textit{DataPort}(n', \textit{port})) = \textit{pipe.alphabet}) \\
\\
\forall\, n : \textit{CONNNAME};\ n' : \textit{COMPNAME};\ \textit{port} : \textit{PORT} \\
\bullet\ \textit{attachment}(n, \textit{reader}) = (n', \textit{port}) \Rightarrow \\
\qquad (\exists\, \textit{fil} : \mathcal{M}^{PF}_{Comp}(\textit{components}(n')); \\
\qquad\qquad \textit{pipe} : \mathcal{M}^{PF}_{Conn}(\textit{connectors}(n)) \\
\qquad\qquad \bullet\quad \textit{DataPort}(n', \textit{port}) \in \textit{fil.inputs} \\
\qquad\qquad\quad \wedge \textit{fil.alphabet}(\textit{DataPort}(n', \textit{port})) = \textit{pipe.alphabet}) \\
\hline
\end{array}
$$

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*.

$$\operatorname{dom} \mathcal{M}^{PF}_{Conf} = \textit{LegalPFConfig}$$

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[9, 16].

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 7 gives an overview of the event system architectural 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.

$$[\textit{METHOD}, \textit{EVENT}]$$

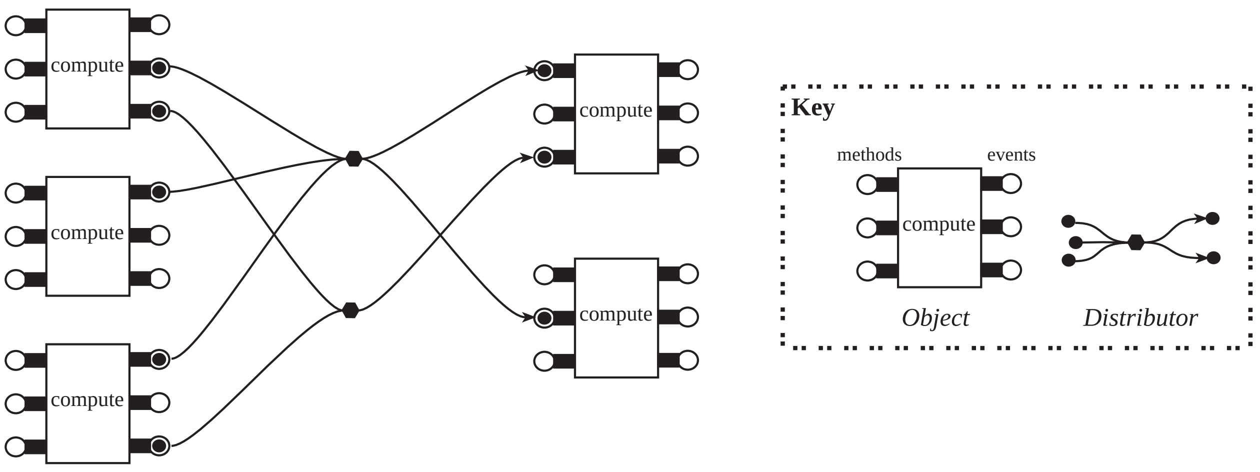

Figure 7: The event system style

$$
\begin{array}{|l}
\textit{Object} \\
\hline
methods : \mathbb{P}\, METHOD \\
events : \mathbb{P}\, EVENT \\
states : \mathbb{P}\, STATE \\
start : STATE \\
transitions : (METHOD \times STATE) \mathrel{+\!\!\!\rightarrow} (STATE \times \mathbb{P}\, EVENT) \\
\hline
start \in states \\
\operatorname{dom} transitions = methods \times states \\
\operatorname{ran} transitions \subseteq \{\, s : states;\ es : \mathbb{P}\, EVENT \bullet (s, es) \,\} \\
\hline
\end{array}
$$

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.

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

A collection of objects and distributors are joined to form a set of interacting objects. The overall binding of methods to events which is supported by the set of interaction objects 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 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 targetted to some destination outside the system.

InteractingObjectSet

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

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

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

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 $\uminus$) and the announced events are suitably augmented (using the bag union operator $\uplus$).

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

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

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

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.

$$\textit{IO\_Step} \mathrel{\widehat{=}} \textit{IO\_ObjectStep} \lor \textit{IO\_DistributorStep}$$

5.2 Concrete Syntax

A concrete syntax for events systems can be developed as an extension of regular programming languages [19]. The details of these extensions are not particularly important for this discussion. These concrete descriptions define a subset of allowable computation and communication descriptions.

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

For example, Figure 8 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 [10].

```
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 8: 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.

$$
\begin{array}{|l}
EventasPort : PortInst \rightarrowtail\!\!\!\!\!+\; EVENT \\
MethodasPort : PortInst \rightarrowtail\!\!\!\!\!+\; METHOD \\
\hline
\langle \mathrm{dom}\, EventasPort, \mathrm{dom}\, MethodasPort \rangle \text{ partition } PortInst \\
\\
\forall\, n, n' : COMPNAME;\ p : PORT \\
\bullet\quad (n, p) \in \mathrm{dom}\, EventasPort \\
\qquad \Leftrightarrow (n', p) \in \mathrm{dom}\, EventasPort \\
\quad \wedge (n, p) \in \mathrm{dom}\, MethodasPort \\
\qquad \Leftrightarrow (n', p) \in \mathrm{dom}\, MethodasPort
\end{array}
$$

The ES style interprets components as (equivalence classes of) objects, matching the methods and events of the object to corresponding port instances.

$$
\begin{array}{|l}
\mathcal{M}^{ES}_{Comp} : Component \rightarrowtail\!\!\!\!\!+\; \mathbb{P}\, Object \\
\hline
\forall\, c : Component;\ o_1, o_2 : Object \\
\qquad \mid o_1 \in \mathcal{M}^{ES}_{Comp}(c) \\
\qquad \bullet\ o_2 \in \mathcal{M}^{ES}_{Comp}(c) \Leftrightarrow o_1 \equiv_{obj} o_2 \\
\\
\forall\, n : COMPNAME;\ c : \mathrm{dom}\, \mathcal{M}^{ES}_{Comp} \\
\qquad \bullet\ \exists\, o : \mathcal{M}^{ES}_{Comp}(c) \\
\qquad\qquad \bullet\ EventasPort^{\sim} (\!| o.events |\!) \cup MethodasPort^{\sim} (\!| o.methods |\!) \\
\qquad\qquad\qquad = \{\, n \,\} \times c.ports
\end{array}
$$

The ES style interprets connectors as distributors. Roles are identified as either event roles or method roles. The distributor represented must have the same number of events and methods as the connector has roles. Note that we are essentially defining the criteria for equivalence of distributors.

$$
\begin{array}{|l}
EventRoles : \mathbb{P}\, ROLE \\
MethodRoles : \mathbb{P}\, ROLE \\
\hline
\langle EventRoles, MethodRoles\rangle \ \mathsf{partition}\ ROLE
\end{array}
$$

$$
\begin{array}{|l}
\mathcal{M}^{ES}_{Conn} : Connector \rightarrowtail\!\!\!\!\!\!\!+\ \mathbb{P}\, Distributor \\
\hline
\forall\, c : Connector;\ d : Distributor \\
\quad \mid d \in \mathcal{M}^{ES}_{Conn}(c) \\
\quad \bullet \quad \#(d.events) = \#(c.roles \cap EventRoles) \\
\quad\quad \wedge \quad \#(d.methods) = \#(c.roles \cap MethodRoles)
\end{array}
$$

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.

$$
\begin{array}{|l}
\mathcal{M}^{ES}_{Conf} : Configuration \mathrel{+\!\!\!\!\rightarrow} InteractingObjectSet \\
\hline
\forall\, cfg : \mathrm{dom}\, \mathcal{M}^{ES}_{Conf} \bullet \\
(\mathcal{M}^{ES}_{Conf}(cfg)).objects = \\
\{ n : \mathrm{dom}\, cfg.components;\ c : Component;\ o : Object \\
\quad \mid \quad cfg.components(n) = c \\
\quad\quad \wedge o \in \mathcal{M}^{ES}_{Comp}(c) \\
\quad\quad \wedge EventasPort^{\sim} (\!| o.events |\!) \cup MethodasPort^{\sim} (\!| o.methods |\!) \\
\quad\quad\quad = \{n\} \times c.ports \\
\quad \bullet\ o \} \\
\wedge \\
(\mathcal{M}^{ES}_{Conf}(cfg)).distributors = \\
\{ n : \mathrm{dom}\, cfg.connectors;\ c : Connector;\ d : Distributor \\
\quad \mid \quad cfg.connectors(n) = c \\
\quad\quad \wedge d \in \mathcal{M}^{ES}_{Conn}(c) \\
\quad\quad \wedge (\forall\, r : c.roles;\ (n', p) \in \mathrm{dom}\, EventasPort \\
\quad\quad\quad \bullet\ cfg.attachment(n, r) = (n', p) \Leftrightarrow \\
\quad\quad\quad\ \ EventasPort(n', p) \in d.events) \\
\quad\quad \wedge (\forall\, r : c.roles;\ (n', p) \in \mathrm{dom}\, MethodasPort \\
\quad\quad\quad \bullet\ cfg.attachment(n, r) = (n', p) \Leftrightarrow \\
\quad\quad\quad\ \ MethodasPort(n', p) \in d.methods) \\
\quad \bullet\ d \}
\end{array}
$$

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*.

$$
\begin{array}{|l|}
\hline
LegalObject \\
Component \\
\hline
description \in ObjectDescriptions \\
\hline
\end{array}
$$

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

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{LegalDistributor}} \\
\textit{Connector} \\
\hline
\textit{description} \in \textit{DistributorDescriptions} \\
\hline
\end{array}
$$

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.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{LegalESConfig}} \\
\textit{Configuration} \\
\hline
\forall\, c : \operatorname{ran} \textit{components} \bullet c \in \textit{LegalObject} \\
\forall\, c : \operatorname{ran} \textit{connectors} \bullet c \in \textit{LegalDistributor} \\
\\
\forall\, n : \textit{CONNNAME};\ m : \textit{COMPNAME}; \\
\qquad\quad \textit{role} : \textit{ROLE};\ \textit{port} : \textit{PORT} \\
\quad\ \mid ((n, \textit{role}), (m, \textit{port})) \in \textit{attachment} \\
\quad\ \bullet\ \ \textit{role} \in \textit{EventRoles} \Leftrightarrow \\
\qquad\qquad (m, \textit{port}) \in \operatorname{dom} \textit{EventasPort} \\
\qquad\ \ \wedge \textit{role} \in \textit{MethodRoles} \Leftrightarrow \\
\qquad\qquad (m, \textit{port}) \in \operatorname{dom} \textit{MethodasPort} \\
\\
\forall\, \textit{cn} : \operatorname{dom} \textit{connectors};\ r : \textit{connectors}(\textit{cn}).\textit{roles} \\
\quad\ \bullet (\textit{cn}, r) \in \operatorname{dom} \textit{attachment} \\
\hline
\end{array}
$$

The domains of the meaning functions are accordingly defined.

$$
\begin{aligned}
&\operatorname{dom} \mathcal{M}^{ES}_{Comp} = \textit{LegalObject} \\
&\operatorname{dom} \mathcal{M}^{ES}_{Conn} = \textit{LegalDistributor} \\
&\operatorname{dom} \mathcal{M}^{ES}_{Conf} = \textit{LegalESConfig}
\end{aligned}
$$

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:

- disallowing feedback loops, or cycles;
- restriction to a pipeline; and
- allowing only a fan-out of 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.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{PFGraph}} \\
\hline
\textit{LegalPFConfig} \\
\textit{connect} : \textit{COMPNAME} \leftrightarrow \textit{COMPNAME} \\
\hline
\textit{connect} = \\
\qquad \{(c_1, p_1), (c_2, p_2) : \textit{PortInst}; \textit{pipe} : \operatorname{dom} \textit{connectors} \\
\qquad \mid \quad \textit{attachment}(\textit{pipe}, \textit{writer}) = (c_1, p_1) \\
\qquad \quad \wedge \textit{attachment}(\textit{pipe}, \textit{reader}) = (c_2, p_2) \\
\qquad \bullet (c_1, c_2)\} \\
\hline
\end{array}$$

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

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AcyclicPF}} \\
\hline
\textit{PFGraph} \\
\hline
\operatorname{id} \textit{COMPNAME} \cap \textit{connect}^{+} = \varnothing \\
\hline
\end{array}$$

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.

$$\begin{array}{|l}
\mathcal{M}^{Acyclic}_{Comp} : \textit{Component} \rightarrow\!\!\!\!\!+ \mathbb{P}\,\textit{Filter} \\
\mathcal{M}^{Acyclic}_{Conn} : \textit{Connector} \rightarrow\!\!\!\!\!+ \mathbb{P}\,\textit{Pipe} \\
\mathcal{M}^{Acyclic}_{Conf} : \textit{Configuration} \rightarrow\!\!\!\!\!+ \textit{InteractingFilterSet} \\
\hline
\mathcal{M}^{Acyclic}_{Comp} = \mathcal{M}^{PF}_{Comp} \\
\mathcal{M}^{Acyclic}_{Conn} = \mathcal{M}^{PF}_{Conn} \\
\mathcal{M}^{Acyclic}_{Conf} = \{\textit{Acyclic} \bullet \theta\textit{Configuration}\} \lhd \mathcal{M}^{PF}_{Conf}
\end{array}$$

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.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Pipeline}} \\
\hline
\textit{PFGraph} \\
\hline
\exists\, \textit{filters} : \operatorname{seq} \textit{COMPNAME} \\
\qquad \mid \operatorname{ran} \textit{filters} = \operatorname{dom} \textit{components} \\
\qquad \bullet\ \textit{connect} = \quad \{i : 1 \,..\, (\#\textit{filters} \Leftrightarrow 1) \\
\qquad\qquad\qquad\qquad \bullet (\textit{filters}(i), \textit{filters}(i+1))\} \\
\hline
\end{array}$$

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.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{FanOut}} \\
\hline
\textit{PFGraph} \\
\hline
\textit{connect}^{\sim} \in \textit{COMPNAME} \rightarrow\!\!\!\!\!+ \textit{COMPNAME} \\
\hline
\end{array}$$

Garlan and Notkin have used the event system model to investigate the differences between various implementations of an implicit invocation mechanism [9]. 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.

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

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.

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

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.

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

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

$$
\begin{array}{|l}
\textit{AcyclicArch} \\
\hline
\textit{ArchGraph} \\
\hline
\operatorname{id} \textit{COMPNAME} \cap \textit{connect}^{+} = \varnothing \\
\hline
\end{array}
$$

The acyclic event system is easily derived from this.

$$\textit{AcyclicES} \mathrel{\widehat{=}} \textit{ESGraph} \wedge \textit{AcyclicArch}$$

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 here how to do the latter option.

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

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

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 [2]. 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.

$$\mid\ \_\,\textit{collapse}_{PF}\,\_ : \textit{InteractingFilterSet} \leftrightarrow \textit{Filter}$$

This result means 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

$$\mid\ \_\,\textit{collapse}_{ES}\,\_ : \textit{InteractingObjectSet} \leftrightarrow \textit{Object}$$

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. 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

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. This research was sponsored by the National Science Foundation under Grant Number CCR-9112880 and by Siemens Corporate Research. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government or of Siemens Corporation.

References

[1] 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.

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

[3] Allen, R., and Garlan, D. Formalizing architectural connection. In *International Conference on Software Engineering: ICSE-16* (Sorrento, Italy, May 1994), ???, p. ???

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

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

[6] 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.

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

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

[9] 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.

[10] 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).

[11] 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.

[12] Horowitz, B. M. The importance of architecture in DOD software. Tech. Rep. M91-35, The MITRE Corporation, July 1991.

[13] Luckham, D. C., and Vera, J. Event-based concepts and language for system architecture. Working draft, October 1992.

[14] 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.

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

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

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

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

[19] 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.

A 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 [18].

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 $X \times Y$.
  - $X \rightarrowtail\!\!\!\!\!+ Y$ 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.
  - $X \longrightarrow Y$ is the set of all total functions. Total functions are defined on all elements of the domain type.
  - $X \rightarrowtail\!\!\!\!+ Y$ 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).
  - $X \rightarrowtail Y$ denotes the total injective functions from X to Y.
  - $X \rightarrowtail\!\!\!\!\twoheadrightarrow Y$ 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 $1 \mathrel{..} n$ 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

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

$authors = \operatorname{dom} index$

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

$authors = \operatorname{dom} index$

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 $X \nrightarrow Y$, then $f \oplus g$ is another function of type $X \nrightarrow Y$ 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*.