Formalizing Style to Understand Descriptions of Software Architecture

Gregory Abowd* Robert Allen David Garlan

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

July 21, 1995

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.

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 [16, 26]. 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 [16]. 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?

*Gregory Abowd's current address is College of Computing, Georgia Institute of Technology, Atlanta, Georgia.

- 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 answer these questions directly: most diagrammatic notations are not sufficiently expressive to serve as a complete architectural specification. 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 affect 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 of 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 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. In effect, the model provides the additional semantic details not present in the diagrammatic representations.

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

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

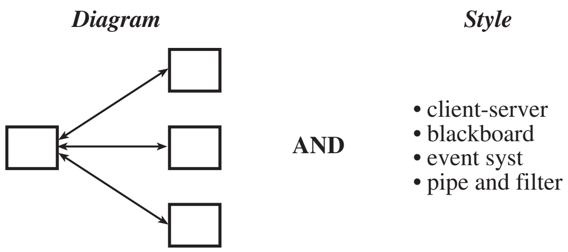

Figure 1: How style distinguishes similar descriptions

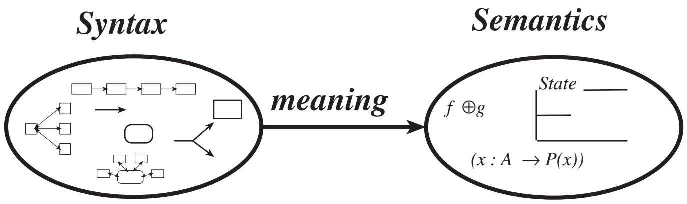

Figure 2: Approach to formalizing architectural style

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 [25], 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, because by convention 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 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 individual styles can be analyzed and 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, 24]. However, it is only recently that software architecture has begun to emerge as a 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 [29] and later Garlan and Shaw [16] extended this list, providing several examples of their use in understanding real systems. Concurrently, Perry and Wolf [26] 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 design patterns. Inspired, in part, by Christopher Alexander's work on pattern languages [1], these efforts have led to handbooks of common patterns for organizing software [10, 27]. The patterns usually consist of a small number of objects that interact in specific ways.

Our research builds on such 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 [31].

Languages for architectural description

In an attempt to provide architectural design with better notations, several new languages have been proposed for architectural description. Rapide [20] provides a module description language, whose interface model is based on events and event patterns. UniCon [30] provides an architectural description language in which both components and connectors have interfaces and can be associated with implementations. Wright [5] is an architectural specification language that allows one to define the semantics of connectors as formal protocols in a variant of CSP [18]. 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, 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 [11]. Aesop was specifically designed to support the definition of architectural styles. New styles are defined as 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, 22]. 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 at 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 [17, 34].

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 [12], a pipe-filter style [2], and an implicit invocation style [14]. 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 characterized styles as theories in first order predicate logic. [23]. 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. We also focus more on the analysis of properties of styles than on the question of refinement between styles.

Inverardi and Wolf have used the Chemical Abstract Machine [6] as a formal basis for architectural description [19]. 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.

In their work on architectures for distributed systems, Magee and Kramer have used the π-calculus to model the dynamic aspects of architectures described in the Darwin language [21]. Their work can be viewed as a good example of formalization for a particular style (embodied in Darwin) in a semantic model different than the one we use in this paper.

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. The significance of the method defined in Section 2 is that it provides a uniform approach to defining any architectural style and for reasoning within and between styles. To demonstrate this, in the following sections we define two quite different architectural styles and show examples of analyses for each. The key points are that there is a uniform way to give semantics to styles and that such formalisms can support useful formal analyses.

Before defining the two styles in Section 3, we first abstract and formalize the concepts behind the box-and-line diagrams that are prevalent in current informal architectural descriptions as a *syntactic domain*. This portion of our formalism is style-independent. In Section 4 we define a pipe-filter (PF) style. Then in Section 5 we show how specific substyles of PF emerge from that definition. We also outline two non-trivial analyses to show how the formalism can be used to reason about the properties of styles. The first analysis shows that PF supports hierarchical decomposition of components into sub-PF systems. The second examines the question of implementing the general PF model using finitely-buffered pipes, and provides sufficient conditions for doing this. A second style—an implicit invocation event system (ES)—is the subject of Section 6. Our discussion of ES will be somewhat shorter than PF, but we will outline how it is defined and do similar substyle and hierarchical analysis of it in order to demonstrate the leverage gained through the formal framework.

Throughout the paper, we use the Z specification language to describe the formal model. Appendix A summarizes most of the Z notation used in this paper. For additional details on Z, see [32]. 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 suffice 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 and/or control 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 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.

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

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 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

[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.

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

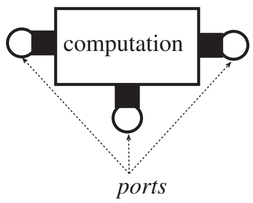

Figure 3: A component and a connector

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. 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

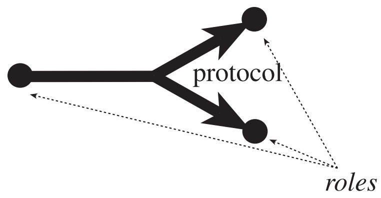

Figure 4: A connector

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 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*.

$[ROLE, CONNDESC]$

[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.

$$
\begin{array}{|l}
\hline
\textit{Connector} \\
\textit{roles} : \mathbb{P}\, \textit{ROLE} \\
\textit{description} : \textit{CONNDESC} \\
\hline
\end{array}
$$

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

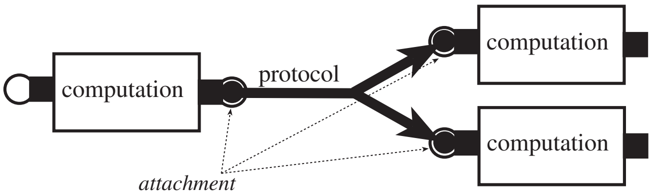

Figure 5: A configuration

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.

$$
\begin{array}{l}
[\textit{COMPNAME}, \textit{CONNNAME}] \\
\textit{PortInst} == \textit{COMPNAME} \times \textit{PORT} \\
\textit{RoleInst} == \textit{CONNNAME} \times \textit{ROLE}
\end{array}
$$

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.

$$
\begin{array}{|l}
\hline
\textit{Configuration} \\
\textit{components} : \textit{COMPNAME} \pfun \textit{Component} \\
\textit{connectors} : \textit{CONNNAME} \pfun \textit{Connector} \\
\textit{attachment} : \textit{RoleInst} \pfun \textit{PortInst} \\
\hline
\forall\, cn : \textit{CONNNAME};\ r : \textit{ROLE} \mid (cn, r) \in \operatorname{dom} \textit{attachment} \\
\quad \bullet\ cn \in \operatorname{dom} \textit{connectors} \land r \in (\textit{connectors}(cn)).\textit{roles} \\
\\
\forall\, cn : \textit{COMPNAME};\ p : \textit{PORT} \mid (cn, p) \in \operatorname{ran} \textit{attachment} \\
\quad \bullet\ cn \in \operatorname{dom} \textit{components} \land p \in (\textit{components}(cn)).\textit{ports} \\
\hline
\end{array}
$$

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.

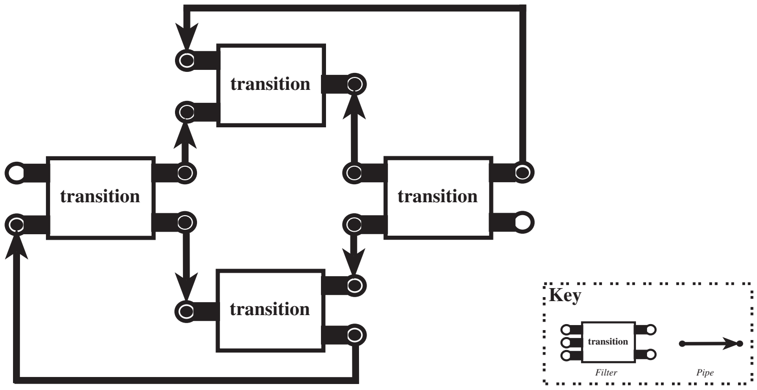

Figure 6: An instance of the pipe-filter style

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. At this level of abstraction we are not interested in the representational details of the internal state and data, 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).

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{Filter}}} \\
\mathit{inputs}, \mathit{outputs} : \mathbb{P}\, \mathit{DATAPORT} \\
\mathit{alphabet} : \mathit{DATAPORT} \pfun \mathbb{P}\, \mathit{DATA} \\
\mathit{states} : \mathbb{P}\, \mathit{STATE} \\
\mathit{start} : \mathit{STATE} \\
\mathit{transitions} : (\mathit{STATE} \times (\mathit{DATAPORT} \pfun \operatorname{seq} \mathit{DATA})) \\
\qquad\qquad \leftrightarrow (\mathit{STATE} \times (\mathit{DATAPORT} \pfun \operatorname{seq} \mathit{DATA})) \\
\hline
\mathit{inputs} \cap \mathit{outputs} = \varnothing \\
\operatorname{dom} \mathit{alphabet} = \mathit{inputs} \cup \mathit{outputs} \\
\mathit{start} \in \mathit{states} \\
\\
\forall s_1, s_2 : \mathit{STATE}; ps_1, ps_2 : \mathit{DATAPORT} \pfun \operatorname{seq} \mathit{DATA} \\
\qquad \bullet ((s_1, ps_1), (s_2, ps_2)) \in \mathit{transitions} \Rightarrow \\
\qquad\qquad\qquad s_1 \in \mathit{states} \land s_2 \in \mathit{states} \\
\qquad\qquad \land \quad \operatorname{dom} ps_1 = \mathit{inputs} \land \operatorname{dom} ps_2 = \mathit{outputs} \\
\qquad\qquad \land \quad (\forall i : \mathit{inputs} \bullet \operatorname{ran}(ps_1(i)) \subseteq \mathit{alphabet}(i)) \\
\qquad\qquad \land \quad (\forall o : \mathit{outputs} \bullet \operatorname{ran}(ps_2(o)) \subseteq \mathit{alphabet}(o)) \\
\hline
\end{array}
$$

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).

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{FilterState}}} \\
f : \mathit{Filter} \\
\mathit{curstate} : \mathit{STATE} \\
\mathit{instate}, \mathit{outstate} : \mathit{DATAPORT} \pfun \operatorname{seq} \mathit{DATA} \\
\hline
\mathit{curstate} \in f.\mathit{states} \\
\operatorname{dom} \mathit{instate} = f.\mathit{inputs} \\
\operatorname{dom} \mathit{outstate} = f.\mathit{outputs} \\
\forall p : f.\mathit{inputs} \bullet \operatorname{ran}(\mathit{instate}(p)) \subseteq f.\mathit{alphabet}(p) \\
\forall p : f.\mathit{outputs} \bullet \operatorname{ran}(\mathit{outstate}(p)) \subseteq f.\mathit{alphabet}(p) \\
\hline
\end{array}
$$

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 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).

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad FilterStep \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
\Delta FilterState \\
\hline
f' = f \\
\\
\exists\, in, out : DATAPORT \nrightarrow \mathrm{seq}\ DATA \bullet \\
\qquad\quad ((curstate, in), (curstate', out)) \in f.transitions \\
\qquad \wedge\ \forall\, p : f.inputs \bullet instate(p) = in(p) \frown instate'(p) \\
\qquad \wedge\ \forall\, p : f.outputs \bullet outstate'(p) = outstate(p) \frown out(p) \\
\hline
\end{array}
$$

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.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Pipe \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
source, sink : DATAPORT \\
alphabet : \mathbb{P}\ DATA \\
\hline
source \neq sink \\
\hline
\end{array}
$$

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.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad PipeState \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
p : Pipe \\
sourcedata : \mathrm{seq}\ DATA \\
sinkdata : \mathrm{seq}\ DATA \\
\hline
\mathrm{ran}\ sourcedata \subseteq p.alphabet \\
\mathrm{ran}\ sinkdata \subseteq p.alphabet \\
\hline
\end{array}
$$

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.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad PipeStep \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
\Delta PipeState \\
\hline
p = p' \\
\\
\exists\, deliver : \mathrm{seq}\ DATA \mid \#deliver > 0 \bullet \\
\qquad (deliver \frown sourcedata' = sourcedata) \wedge (sinkdata' = sinkdata \frown deliver) \\
\hline
\end{array}
$$

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.

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

The behavior of an interacting set of filters is defined as the behaviors of the constituent filters and pipes. The state of the system identifies filter and pipe states with filters and pipes in the system.

$$
\begin{array}{l}
\textit{PFSystemState} \\
\hline
\textit{sys} : \textit{InteractingFilterSet} \\
\textit{filterstates} : \mathbb{P}\,\textit{FilterState} \\
\textit{pipestates} : \mathbb{P}\,\textit{PipeState} \\
\hline
\textit{sys}.\textit{filters} = \{\textit{fs} : \textit{filterstates} \bullet \textit{fs}.f\} \\
\forall \textit{fs}_1, \textit{fs}_2 : \textit{filterstates} \bullet \textit{fs}_1.f = \textit{fs}_2.f \Leftrightarrow \textit{fs}_1 = \textit{fs}_2 \\
\\
\textit{sys}.\textit{pipes} = \{\textit{ps} : \textit{pipestates} \bullet \textit{ps}.p\} \\
\forall \textit{ps}_1, \textit{ps}_2 : \textit{pipestates} \bullet \textit{ps}_1.p = \textit{ps}_2.p \Leftrightarrow \textit{ps}_1 = \textit{ps}_2
\end{array}
$$

A step in this behavior is either a computation step for one filter or a transmission step for one pipe, all else remaining unchanged. The definition of a system-level filter or pipe computation step are not given here. Those definitions are not difficult, but the framing conditions stating that all else remains unchanged are somewhat cumbersome. Full details of these specifications are given in [3]. We summarize here by limiting a system computation to only an individual filter or pipe computation.

$$\textit{PFSystemStep} \mathrel{\widehat{=}} \textit{SystemFilterStep} \vee \textit{SystemPipeStep}$$

A legal *trace* of the system is a sequence of computation steps. (We will need this definition of traces in our analysis of the PF style later in Section 5.)

$$
\begin{array}{l}
\mathcal{T}^{PF} : \textit{InteractingFilterSet} \longrightarrow \mathbb{P}(\mathrm{seq}\ \textit{PFSystemState}) \\
\hline
\forall s : \textit{InteractingFilterSet} \bullet \\
\textit{PFtraces}(s) = \{\textit{trace} : \mathrm{seq}\ \textit{PFSystemState} \mid \\
\qquad \textit{trace}(1).\textit{sys} = s \wedge \\
\qquad \forall i : 1..(\#\textit{trace} - 1) \\
\qquad \bullet (\exists \textit{PFSystemState}; \textit{PFSystemState}' \\
\qquad\qquad \bullet \theta \textit{PFSystemState} = \textit{trace}(i) \\
\qquad\qquad \wedge\ \theta \textit{PFSystemState}' = \textit{trace}(i+1) \\
\qquad\qquad \wedge\ \textit{PFSystemStep} \\
\qquad\qquad \wedge\ \textit{sys} = s) \\
\qquad \}
\end{array}
$$

```
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

$$\mathcal{T}^{PF} : \mathit{InteractingFilterSet} \longrightarrow \mathbb{P}(\mathrm{seq}\ \mathit{PFSystemState})$$

$$\begin{aligned}
&\forall\, s : \mathit{InteractingFilterSet} \bullet \mathcal{T}^{PF}(s) = \\
&\quad \{\mathit{trace} : \mathrm{seq}\ \mathit{PFSystemState} \mid \forall\, i : 1..(\#\mathit{trace} - 1) \bullet (\exists\, \mathit{PFSystemState};\ \mathit{PFSystemState}' \\
&\qquad \bullet \theta \mathit{PFSystemState} = \mathit{trace}(i) \\
&\qquad \wedge\ \theta \mathit{PFSystemState}' = \mathit{trace}(i+1) \\
&\qquad \wedge\ \mathit{PFSystemStep} \\
&\qquad \wedge\ \mathit{sys} = s)\}
\end{aligned}$$

4.2 Concrete Syntax

The second part of a style definition is the creation of a style-specific concrete syntax. This part of the definition augments the basic graphical depiction, typically with syntactic information not easily representable diagrammitically.

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.

$$\begin{aligned}
&\mathit{FilterDescriptions} : \mathbb{P}\ \mathit{COMPDESC} \\
&\mathit{PipeDescriptions} : \mathbb{P}\ \mathit{CONNDESC}
\end{aligned}$$

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:

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

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. (The underscores indicate that the defined operator is an infix operator.)

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

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.

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

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.

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

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.

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

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

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.)

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{LegalPFComponent}}} \\
\mathit{Component} \\
\hline
\mathit{description} \in \mathit{FilterDescriptions} \\
\hline
\end{array}
$$

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*.

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

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 show that given any legal PF component, we can apply $\mathcal{M}^{PF}_{Comp}$ to obtain a filter. That is, we must show

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

This amounts to demonstrating that

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

or, in essence, that the function *DataPort* is reasonably constructed and that 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, *reader* and *writer*.

$$
\begin{array}{|l}
\hline
\textit{LegalPFConnector} \\
\textit{Connector} \\
\hline
\textit{description} \in \textit{PipeDescriptions} \\
\textit{roles} = \{\textit{reader}, \textit{writer}\} \\
\hline
\end{array}
$$

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

$$\operatorname{dom} \mathcal{M}^{PF}_{Conn} = \textit{LegalPFConnector}$$

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.

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

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 the formal definition of the PF style. We will now see how we can use this definition to analyze the style.

5 Analyzing the PF style

Although there is direct value in clarifying our intuitions about an architectural style by providing an unambiguous model of it, an additional reason to formalize architectural style is to support reasoning about properties of the style. In this section we present three representative examples of analysis that are supported by our formal framework. First, we show how to relate a style to its "substyles" through incremental syntactic restrictions on the domain of the meaning functions. Second, we can investigate important architectural properties by reasoning about the semantic model. Our example will show that the PF style supports hierarchical decomposition. Third, we will consider the issue of when a general PF system can be implemented using finitely-buffered pipes.

5.1 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.

To address such topological restrictions we consider a PF configuration as a directed graph in whicn two components are connected if any of their ports are attached to a common pipe.

PFGraph

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

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

Acyclic

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

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 before.

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

A *pipeline* architecture is a PF system in which the connection graph is a linear sequence of filters.

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

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.

$$
\begin{array}{|l}
\hline
\textit{FanOut} \\
\textit{PFGraph} \\
\hline
\textit{connect}^{\sim} \in \textit{COMPNAME} \nrightarrow \textit{COMPNAME} \\
\hline
\end{array}
$$

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

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.

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

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

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

The acyclic PF substyle is easily derived from this.

$$\textit{AcyclicPF} \mathrel{\widehat=} \textit{PFGraph} \land \textit{AcyclicArch}$$

5.2 Hierarchical Decomposition

One desirable property of an architectural description is *encapsulation*: components (or connectors) may themselves be represented hierarchically as an architectural configuration. By defining a style formally, we can investigate the properties of a style make it possible to encapsulate a configuration as a higher-level entity in that style.

For PF, it would seem intuitively plausible that a configuration of pipes and filters can be bundled up as another semantically-equivalent filter. But what exactly does this mean? While it is relatively obvious what is involved at the syntactic, diagrammatic level, it is much less clear how to understand the issue at a deeper, semantic level. In this section we provide one answer. In particular, we use the formal model to explain at a semantic level what is meant by "equivalence" between a filter and a configuration, and show that we can always find an equivalent filter for any configuration.

The basic idea behind equivalence is that a system's computations are equivalent to that of a single filter if there is a correspondence between the "externally visible" parts of the system's traces and the filter's traces. The externally visible parts of a system state are the data on the inputs and outputs of filters that are not attached to any pipe. To get things started, we need to refer to the ports on the filters in a configuration that are not attached to any pipe.

$$
\begin{array}{|l}
\mathit{UnBoundPorts} : \mathit{InteractingFilterSet} \mathrel{+\!\!\!\rightarrow} \mathbb{P}\,\mathit{DATAPORT} \\
\hline
\forall\, s : \mathit{InteractingFilterSet} \bullet \\
\mathit{UnboundPorts}(s) = \\
\quad \{f : s.\mathit{filters}, dp : \mathit{DATAPORT} \mid \\
\qquad dp \in (f.\mathit{inputs} \cup f.\mathit{outputs}) \wedge \neg\,(\exists\, p : s.\mathit{pipes} \bullet dp = p.\mathit{sink} \vee dp = p.\mathit{source}) \\
\quad \bullet\, dp\}
\end{array}
$$

Recall the definition of legal traces, $\mathcal{T}^{PF}$ (Section 4.1). We can now extract the *externally observable* traces of a system by projecting out only the states of the unbound ports.

$$
\begin{array}{|l}
\mathit{external} : \mathit{PFSystemState} \longrightarrow (\mathit{DATAPORT} \mathrel{+\!\!\!\rightarrow} \mathrm{seq}\,\mathit{DATA}) \\
\hline
\forall\, ss : \mathit{PFSystemState} \bullet \\
\quad \mathit{external}(ss) = \\
\qquad \{fs : ss.\mathit{filterstates};\ dp : \mathit{DATAPORT} \\
\qquad \mid dp \in \mathit{UnboundPorts}(ss.sys) \wedge dp \in fs.\mathit{filter}.\mathit{inputs} \\
\qquad \bullet\, dp \mapsto fs.\mathit{instate}(dp)\} \\
\qquad \cup \\
\qquad \{fs : ss.\mathit{filterstates};\ dp : \mathit{DATAPORT} \\
\qquad \mid dp \in \mathit{UnboundPorts}(ss.sys) \wedge dp \in fs.\mathit{filter}.\mathit{outputs} \\
\qquad \bullet\, dp \mapsto fs.\mathit{outstate}(dp)\}
\end{array}
$$

$$
\begin{array}{|l}
\mathcal{T}^{PF}_{ext} : \mathit{InteractingFilterSet} \longrightarrow \mathbb{P}(\mathrm{seq}(\mathit{DATAPORT} \mathrel{+\!\!\!\rightarrow} \mathrm{seq}\,\mathit{DATA})) \\
\hline
\forall\, s : \mathit{InteractingFilterSet};\ t : \mathcal{T}^{PF}(s) \bullet t \mathbin{;} \mathit{external} \in \mathcal{T}^{PF}_{ext}(s)
\end{array}
$$

We can now define what it means for two computations to be equivalent:

$$
\begin{array}{|l}
\mathit{equiv}_{PF} : \mathit{InteractingFilterSet} \longleftrightarrow \mathit{InteractingFilterSet} \\
\hline
\mathit{equiv}_{PF} = \{sys_1, sys_2 : \mathit{InteractingFilterSet} \bullet \mathcal{T}^{PF}_{ext}(sys_1) = \mathcal{T}^{PF}_{ext}(sys_2)\}
\end{array}
$$

In order to address the equivalence of a system with a single filter, it is simplest to view the filter as a PF system containing exactly that filter and no pipes:

$$
\begin{array}{|l|}
\hline
\mathit{SingleFilter} \\
\hline
\mathit{InteractingFilterSet} \\
\hline
\#\mathit{filters} = 1 \wedge \#\mathit{pipes} = 0 \\
\hline
\end{array}
$$

The encapsulation property can now be stated formally: For any set of interacting filters there is a corresponding single filter that has equivalent externally observable traces:

$$\forall\, sys : InteractingFilterSet \bullet$$
$$\quad \exists\, fil : SingleFilter \bullet (sys, fil) \in equiv_{PF}$$

The proof of the theorem proceeds by induction. There are two simple base cases—a system with no filters, and a system with exactly one filter. The latter case is trivial, and the former follows from the existence of a filter with no inputs and no outputs.

The first induction case is for a system with no pipes:

$$\#sys.filters = n + 1 \wedge \#sys.pipes = 0$$

We divide the system into two parts, a system with n filters, sys_n, and a single filter f. By the induction hypothesis, the sys_n can be encapsulated as a filter f_n. It remains to contruct a filter f' that is equivalent to the computation of the two filters f_n and f.

f' is constructed by mapping pairs of states, one each from f and f_n, into a single state using an auxiliary function *fstatefun*:

$$fstatefun : (STATE \times STATE) \rightarrowtail STATE$$

We know that *fstatefun* exists because $STATE$ is a countably infinite set. We can now construct f' as follows:

$$\begin{aligned}
&f'.states = fstatefun(\!| f.states \times f_n.states |\!)\\
&f'.inputs = f.inputs \cup f_n.inputs\\
&f'.outputs = f.outputs \cup f_n.outputs\\
&f'.alphabet = f.alphabet \cup f_n.alphabet
\end{aligned}$$

$$\begin{aligned}
&f'.transitions =\\
&\quad \{\, ((s1, i), (s2, o)) : f.transitions;\ s_n : f_n.states\\
&\qquad \bullet ((fstatefun(s1, s_n), i \cup \{f : f_n.inputs \bullet f \mapsto \langle\rangle\}),\\
&\qquad (fstatefun(s2, s_n), o \cup \{f : f_n.outputs \bullet f \mapsto \langle\rangle\}))\,\}\\
&\quad \cup\\
&\quad \{\, ((s1, i), (s2, o)) : f_n.transitions;\ s : f.states\\
&\qquad \bullet ((fstatefun(s, s1), i \cup \{f : f.inputs \bullet f \mapsto \langle\rangle\}),\\
&\qquad (fstatefun(s, s2), o \cup \{f : f.outputs \bullet f \mapsto \langle\rangle\}))\,\}
\end{aligned}$$

This yields a "union" of the two filters. The *transitions* of f' are either a transition of f_n or of f. A transition is constructed by changing the internal state and ports of one of the f_n or f. The other filter's state is unchanged, its inputs are ignored, and its outputs are left untouched. This is exactly the behavior of a *SystemFilterStep*, and so f' is indeed the required filter.

The second and final induction step handles the addition of pipes to a system:

$$sys.filters = F \wedge \#sys.pipes = n + 1$$

Again, the induction hypothesis ensures that we can divide *sys* into a system sys_n and a single pipe p, such that sys_n has an equivalent filter f_n. We construct a filter f' that is equivalent to a system containing f_n and p. In order to do so, we must encode the internal state of f_n and the full state of p as a single element of internal state.

$$pstatefun : STATE \times \mathrm{seq}\, DATA \times \mathrm{seq}\, DATA \rightarrowtail STATE$$

pstatefun exists because $STATE$ and $DATA$ are both countable (and $\mathrm{seq}\, DATA$ includes only *finite* sequences). We will use *pstatefun* as follows: the $STATE$ represents the state of f_n, the first sequence represents the source side of p (an output of f_n), and the second sequence represents the sink side of p (an input to f_n). The transition function of f' combines transitions of the filter f_n with transmissions of the pipe

p, with all input and output data from the source or sink of p being subsumed into the state of f_n. The construction of f' follows.

$$
\begin{array}{l}
f'.states = pstatefun(\!| f_n.states \times \operatorname{seq} p.alphabet \times \operatorname{seq} p.alphabet |\!) \\
f'.inputs = f_n.inputs \setminus \{p.sink\} \\
f'.outputs = f_n.outputs \setminus \{p.source\} \\
f'.alphabet = \{p.sink, p.source\} \lhd f_n.alphabet \\
f'.transitions = \\
\quad \{((st1, i), (st2, o)) : f_n.transitions;\ d1, d2 : \operatorname{seq} p.alphabet \bullet \\
\qquad ((pstatefun(st1, d1, d2 \frown i(p.sink)), \{p.sink\} \lhd i), \\
\qquad (pstatefun(st2, o(p.source) \frown d1, d2), \{p.source\} \lhd o))\} \\
\quad \cup \\
\quad \{st : f_n.states;\ d1, d2, d : \operatorname{seq} p.alphabet \\
\qquad \bullet ((pstatefun(st, d1 \frown d, d2), (f'.inputs \times \{\varnothing\})), \\
\qquad (pstatefun(st, d1, d \frown d2), (f'.outputs \times \{\varnothing\})))\}
\end{array}
$$

The two induction steps all pipe-filter systems, and so the theorem is proved.

It might seem that this result is so obvious as not to require a proof (or even a formal model and argument). Interestingly, however, our first attempt to define a PF system failed the proof, and led us to revise architectural model. This earlier version of PF modelled the internal computations of a filter as *finite* state machine, although pipes were as above. This led to problems because in an encapsulated PF system, the pipes can retain infinite state, thereby leading to an *infinite* state machine. In Section 6.6 we will see another example where such an encapsulation theorem fails to hold.

5.3 Finitely-Implementable Pipes

As a third example of style analysis, we consider the problem of understanding when a PF system can be efficiently implemented using traditional mechanisms for pipe communication.

An important property of the pipe-filter style, as it has been defined above, is that a pipe-filter computation can use an *infinite* amount of buffer space for its pipes. Consequently there are configurations of filters and pipes that do not fit into any fixed amount of storage space. This may present a practical problem to implementers of systems in the style—particularly if the target implementation uses fixed sized buffers for its pipe implementations (as does Unix, for example). It would thus be valuable for a system developer to have a means of analyzing whether a described system can be implemented using finitely-buffered pipes. In this section we make these notions precise, and show how our architectural framework leads to sufficient conditions under which we can guarantee that a PF system can be implemented in this way.

Definition of Finitely-Implementable Pipe-Filter System

In order to understand the properties of systems that can be implemented using fixed buffering space, we must first understand how buffer space can be consumed. Then we can define what it means for a system to be able to execute in finite space.

We consider a system to be finitely-implementable if all of its computations can be carried out using finite buffers in the pipes. To make this idea precise, we must first clarify which of a system's computations are of interest. We will consider only *terminated* computations. As we will see, restricting the computations to those that have terminated will permit us to restrict the intermediate stages used in a computation without limiting the functional capabilities of a system. Formally, a terminated computation is a system trace where there is no computation step that will extend it. (Again, we use $\mathcal{T}^{PF}(s)$ to refer to the traces of a system s.)

$$
\begin{array}{|l}
\mathit{TerminationsOf} : \mathit{InteractingFilterSet} \longrightarrow \mathbb{P}(\mathrm{seq}\ \mathit{SystemState}) \\
\hline
\forall s : \mathit{InteractingFilterSet} \bullet \mathit{TerminationsOf}(s) = \\
\quad \{t : \mathcal{T}^{PF}(s) \mid \nexists \mathit{SystemComputeStep} \bullet \theta \mathit{SystemState} = t(\#t)\}
\end{array}
$$

We must also define what it means for a computation to be "finite." A finitely-bounded computation is a trace where at no point in the computation does any pipe contain more than a fixed amount of data. (For later ease of manipulation, our definition is parameterized by the actual space bound.)

$$
\begin{array}{|l}
\mathit{FinitelyBoundedTracesOf} : (\mathbb{N} \times \mathit{InteractingFilterSet}) \longrightarrow \mathbb{P}(\mathrm{seq}\ \mathit{SystemState}) \\
\hline
\forall n : \mathbb{N}; s : \mathit{InteractingFilterSet} \bullet \mathit{FinitelyBoundedTracesOf}(n, s) = \\
\quad \{t : \mathit{LegalTracesOf}(s) \mid \forall \mathit{state} : \mathrm{ran}\ t \bullet \forall \mathit{pipeState} : s.\mathit{pipe\_states} \bullet \#\mathit{source\_data} + \#\mathit{sink\_data} \leq n\}
\end{array}
$$

For convenience, we combine these to describe the set of "desirable" computations:

$$
\begin{array}{|l}
\mathit{FiniteTerminationsOf} : (\mathbb{N} \times \mathit{InteractingFilterSet}) \longrightarrow \mathbb{P}(\mathrm{seq}\ \mathit{SystemState}) \\
\hline
\forall n : \mathbb{N}; s : \mathit{InteractingFilterSet} \bullet \mathit{FiniteTerminationsOf}(n, s) = \mathit{FiniteTracesOf}(n, s) \cap \mathit{TerminationsOf}(s)
\end{array}
$$

Given these definitions, we can characterize the criterion a system must meet in order to be implementable using fixed buffer space. A system is finitely-implementable if for every terminating computation of the system there is an equivalent computation that is finitely bounded:

$$
\begin{array}{|l}
\mathit{FinitelyImplementable} : \mathbb{P}(\mathbb{N} \times \mathit{InteractingFilterSet}) \\
\hline
\mathit{FinitelyImplementable} = \{n : \mathbb{N}; s : \mathit{InteractingFilterSet} \mid \\
\quad \forall t : \mathit{TerminationsOf}(s) \bullet \exists t' : \mathit{FiniteTerminationsOf}(n, s) \\
\qquad \bullet\ t(\#t) = t'(\#t') \land t(1) = t'(1)\}
\end{array}
$$

Thus a system S is implementable in fixed space *exactly* when there exists an n such that $\mathit{FinitelyImplementable}(n, S)$.

Analyzing Systems for Finite Implementability

While the definition of finitely-implementable is precise, the definition is not a particularly useful one, since it provides no guidance for determining whether a given system meets the criterion. It is clearly impractical to check most systems by directly considering all of possible traces—as seems to be implied by the *FinitelyImplementable* predicate.

Instead, we would like to provide more structured criteria that can be applied to a system in parts, yet still provide overall system guarantees. In the remainder of this section, we describe such checks, giving rules for filters that will translate to a sufficient condition for complete pipe-filter systems. While the checks themselves are relatively straightforward, formalizing them permits us to argue that together they imply the finite-buffering property.

One reason a system might require an infinite buffer is that the receiving filter might hang on its input, thus allowing an arbitrary amount of data to pile up on the pipe that supplies it. We must therefore use only filters that guarantee to eventually empty out any pipes that provide it input data:

$$
\begin{array}{|l}
\mathit{NoHangFilter} : \mathbb{P}(\mathbb{N} \times \mathit{Filter}) \\
\hline
\mathit{NoHangFilter} = \{n : \mathbb{N}; f : \mathit{Filter} \mid \\
\quad \mathrm{dom} f.\mathit{transitions} = f.\mathit{states} \times \{\mathit{in} : f.\mathit{inputs}; s : \mathrm{seq}\ \mathit{DATA} \\
\qquad \mid \#s = n \land \forall v : \mathrm{ran}\ s \bullet v \in f.\mathit{alphabet}(\mathit{in})\}
\end{array}
$$

The predicate *NoHangFilter* defines those filters that will *always* be able to read when they get enough input.

Another problem can arise if a filter having two or more inputs ports receives an unbounded amount of data at one input while it is blocked on another. This again can result in a pile-up of data in a pipe. One way to avoid this problem, is to require that upstream filters be *balanced* (*i.e.,* that they not deliver large amounts of data on one pipe while ignoring another):[4]

$$
\begin{array}{|l}
\mathit{BalancedFilter} : \mathbb{P}(\mathbb{N} \times \mathit{Filter}) \\
\hline
\mathit{BalancedFilter} = \{ n : \mathbb{N}; f : \mathit{Filter} \mid \\
\quad \forall \mathit{istate}, \mathit{ostate} : \mathit{STATE}; \mathit{istrings}, \mathit{ostrings} : \mathit{DATAPORT} \nrightarrow \operatorname{seq} \mathit{DATA} \\
\qquad \mid ((\mathit{istate}, \mathit{istrings}), (\mathit{ostate}, \mathit{ostrings})) \in f.\mathit{transitions} \\
\qquad \bullet \forall \mathit{in} : f.\mathit{inputs} \bullet \#\mathit{istrings}(\mathit{in}) = n \\
\qquad \wedge \forall \mathit{out} : f.\mathit{outputs} \bullet \#\mathit{ostrings}(\mathit{out}) = n \\
\quad \}
\end{array}
$$

Balanced filters that do not hang, together with an acyclic connection graph, are sufficient to guarantee a finitely implementable system. Formally stated:

$$
\begin{array}{l}
\forall n : \mathbb{N}; s : \mathit{InteractingFilterSet} \\
\quad \mid (s \in \operatorname{ran} \mathcal{M}^{\mathit{Acyclic}}_{\mathit{Conf}}) \wedge (\forall f : s.\mathit{filters} \bullet \mathit{BalancedFilter}(n,f) \wedge \mathit{NoHangFilter}(n,f)) \\
\quad \bullet \mathit{FinitelyImplementable}(n, s)
\end{array}
$$

Notice that what we have done is to localized the checks to individual filters: the theorem states that if all filters have the desired property the system as a whole will be finitely-implementable. Thus the abstract definition has been turned into a usable result.

In outline, the proof of this property is based on the idea that the computations in a trace of a pipe-filter system can be reordered without affecting the final state of the computation. The only restriction on reordering is that if one computation enables another one, then it cannot be moved to after the computation it enables. Given this property, we can reorder the computations of a trace that violates the constraint on buffer size to move computations that reduce buffer size so that they happen before computations that increase buffer size. (Appendix B sketches the proof in more detail.)

6 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, event systems with implicit invocation (ES). The importance of this section is not so much the details of the ES style, but that we are able to define this style in exactly the same way as we did for the PF style. Though it is an important contribution that we are able to formalize one architectural style, such as PF, and reason about its properties, it is far more important that we provide a method for others to define any of a number of interesting architectural styles, and subject those styles to similar analyses. Hence our discussion of ES will not be as thorough as PF in an attempt to highlight for the reader the form of the definition and analysis.

6.1 Event Systems

Event systems are based on the idea that components in a system interact by means of event broadcast: events "announced" by one component can trigger "method" invocations at the interfaces of zero or more other components. Event systems are becoming increasingly important as a flexible tool integration technique, since they allow the implicit invocation of tools when some other tool announces an event[14, 13, 28].

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

[4] Of course, it is not *necessary* for filters to be balanced in order to avoid the problem. For illustrative purposes, in this paper we are only concerned with a set of *sufficient* conditions.

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.

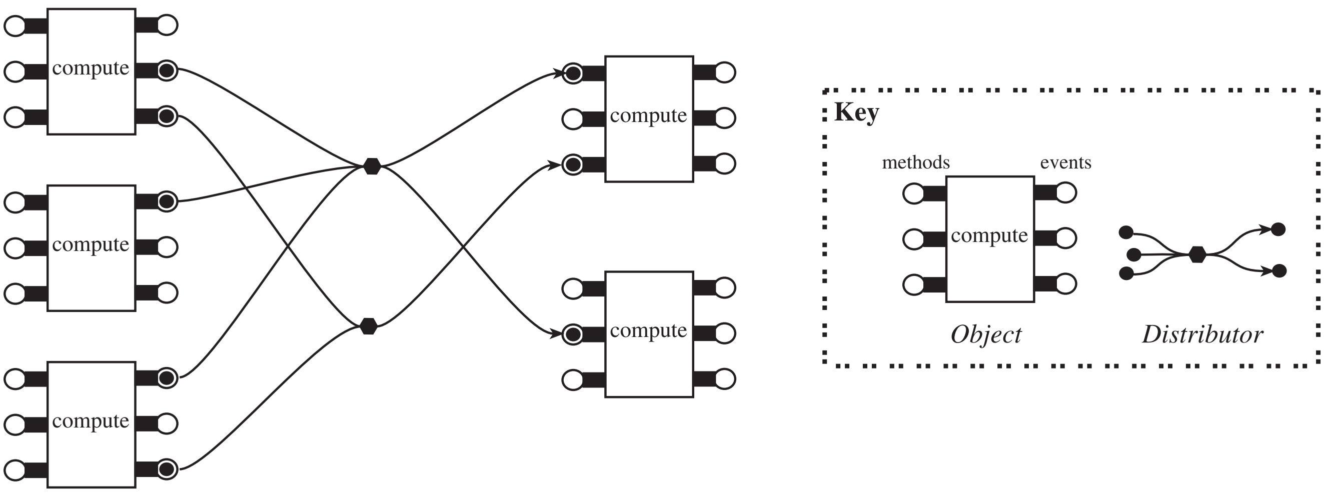

Figure 8: The event system style

6.2 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

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

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

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

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, or multisets (see Appendix A).

IOState

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

We will defer the discussion of the dynamic behavior of ES until later on when we discuss the encapsulation property for this style.

6.3 Concrete Syntax

A concrete syntax for events systems can be developed as an extension of regular programming languages [33]. 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 [15].

6.4 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 $\equiv_{obj}$.

```
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

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.

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

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

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

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 a criteria for equivalence of distributors.

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

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

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. The definition of this mapping function is similar to the one in PF, and we elide it here. (See Appendix C for the details.)

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

$\ldots$

6.5 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

$\mathit{description} \in \mathit{ObjectDescriptions}$

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

$\mathit{description} \in \mathit{DistributorDescriptions}$

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

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

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

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

The domains of the meaning functions are accordingly defined.

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

As before, there is a proof obligation to show that these domain restrictions are sufficient to guarantee the mapping results in semantically legal entities.

6.6 Analysis of ES

As with PF, we now examine some of the analyses that can be applied to ES.

Defining architectural substyles

Garlan and Notkin have used the event system model to investigate the differences between various implementations of an implicit invocation mechanism [14]. 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. (For example, we might insist that at most one object announce a given type of event, or that there are no cycles induced by event-method chaining.) 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. Similar to how we defined a connectivity graph in PF, we can define one for ES as well. Recall that 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. 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.

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

The acyclic substyle of ES is then defined as a syntactic specialization on the more general style.

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

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.

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

Properties of the semantic domain

We proposed the encapsulation property in Section 5.2 and demonstrated that it was supported by the semantic definition for the PF style. We now want to investigate whether encapsulation holds for the semantic domain we have defined for ES. The key to proving this property in the PF example was to determine the behavior of a collection of semantic elements (a set of interacting filters for PF; a set of interacting objects for ES) externally and then decide whether that behavior could be matched by a single element (a filter in PF or an object in ES).

While we were able to prove that this property does hold for PF, it does *not* hold for event systems. In this section we prove a simple atomicity property on the computations of single objects, and show a counter-example that proves this property does not hold of all event systems (i.e., configurations of objects). We thus prove that not all event systems can be encapsulated as single objects.

Before proceeding with the proof, we must precisely define encapsulation for event systems. Recall that encapsulation relies on defining a correspondence between external behavior of a collection of architectural elements and a single element. First, we need to define the externally observable behavior of a set of interacting objects. In the PF example, we defined the external behavior in terms of the unbound input and output ports of the system. For ES, we will define external behavior as methods which are not invoked internally (by some distributor in the encapsulated system) and events which are not translated internally.

$$
\begin{array}{|l}
\mathit{UnboundMethods} : \mathit{InteractingObjectSet} \mathrel{+\!\!\!\rightarrow} \mathbb{P}(\mathit{Object} \times \mathit{METHOD}) \\
\mathit{UnboundEvents} : \mathit{InteractingObjectSet} \mathrel{+\!\!\!\rightarrow} \mathbb{P}(\mathit{Object} \times \mathit{EVENT}) \\
\hline
\forall\, s : \mathit{InteractingObjectSet} \bullet \\
\mathit{UnboundMethods}(s) = \\
\quad \{o : s.\mathit{objects}, m : \mathit{METHOD} \mid m \in (o.\mathit{methods}) \land \neg\, (\exists\, d : s.\mathit{distributors} \bullet m \in d.\mathit{methods}) \\
\qquad \bullet (o, m)\} \\
\mathit{UnboundEvents}(s) = \\
\quad \{o : s.\mathit{objects}, e : \mathit{EVENT} \mid e \in (o.\mathit{events}) \land \neg\, (\exists\, d : s.\mathit{distributors} \bullet e \in d.\mathit{events}) \\
\qquad \bullet (o, e)\}
\end{array}
$$

The externally observed state consists only of the bag of invoked methods and the bag of announced events

$$
\begin{array}{|l}
\mathit{external} : \mathit{IOState} \longrightarrow (\mathrm{bag}\, \mathit{METHOD} \times \mathrm{bag}\, \mathit{EVENT}) \\
\hline
\forall\, ss : \mathit{IOState} \bullet \\
\quad \mathit{external}(ss) = (\mathit{UnboundMethods}(ss.sys) \lhd ss.\mathit{invoked}, \mathit{UnboundEvents}(ss.sys) \lhd ss.\mathit{announced})
\end{array}
$$

The external behavior, denoted by *ESExttraces* extracts the externally observable information from the legal computational traces in $\mathcal{T}^{ES}$.

$$
\begin{array}{|l}
\mathcal{T}^{ES}_{ext} : \mathit{InteractingObjectSet} \longrightarrow \mathbb{P}(\mathrm{seq}(\mathrm{bag}\, \mathit{METHOD} \times \mathrm{bag}\, \mathit{EVENT})) \\
\hline
\forall\, s : \mathit{InteractingObjectSet}; t : \mathrm{seq}\, \mathit{IOState} \bullet \\
\quad t \in \mathcal{T}^{ES}(s) \Leftrightarrow t \mathbin{;} \mathit{external} \in \mathit{ESExttraces}(s)
\end{array}
$$

The equivalence of two event systems can now be defined:

$$
\begin{array}{|l}
\mathit{equiv}_{ES} : \mathit{InteractingObjectSet} \longleftrightarrow \mathit{InteractingObjectSet} \\
\hline
\mathit{equiv}_{ES} = \{sys_1, sys_2 : \mathit{InteractingObjectSet} \mid \mathit{ESExtTraces}(sys_1) = \mathit{ESExtTraces}(sys_2)\}
\end{array}
$$

By equating a single object with a system that contains it (as we did for a single filter), the encapsulation theorem can now be stated for event systems.

$$
\begin{array}{|l}
\hline
\textit{SingleObject} \\
\textit{InteractingObjectSet} \\
\hline
\#objects = 1 \land \#distributors = 0 \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\forall\, sys : InteractingObjectSet \bullet \\
\quad \exists\, obj : SingleObject \bullet (sys, obj) \in equiv_{ES}
\end{array}
$$

This is *false*. To understand why this is so, we must look in more detail at the possible computations a system. The computation of an object in a sysem is defined as follows:

$$
\begin{array}{|l}
\hline
\textit{ObjectComputeStep} \\
\Delta IOState \\
\Xi InteractingObjectSet \\
\hline
\exists\, m : METHOD;\ o : objects \bullet \\
\qquad m \in o.methods \land m \sqsubseteq invoked \\
\qquad \land\ invoked' = invoked \uplus \{m \mapsto 1\} \\
\qquad \land\ announced \sqsubseteq announced' \\
\qquad \land\ ((state(o), m), (state'(o), announced' \uplus announced)) \in o.transitions \\
\qquad \land\ \{o\} \lhd state = \{o\} \lhd state' \\
\hline
\end{array}
$$

A key property of this schema is that the number of methods that have been invoked but not dealt with is reduced by one:

$$
\#invoked' = \#invoked - 1
$$

Because every step in a trace of a *SingleObject* is a computation step of that object, this observation can be translated to a property of the traces of all *SingleObject*s:

$$
\begin{array}{l}
\forall\, so : SingleObject;\ tr : ESExtTraces(so);\ i : 1 \mathrel{..} \#tr - 1 \\
\quad \bullet\ \#first(tr(i)) = \#first(tr(i+1)) + 1
\end{array}
$$

Essentially, this states that all computations of a single object are *atomic*, i.e. all computation is the result of a method invocation and completes in a single step; no future computation can result without another method to trigger it.

Keeping this property in mind, we can now construct a counter-example, sys_{dist}, to our proposed encapsulation theorem:

$$
\begin{array}{|l}
sys_{dist}.objects = \{o1, o2\} \\
sys_{dist}.distributers = \{d\} \\
\hline
\exists\, m1 : o1.methods;\ m2 : o2.methods;\ e1 : o1.events; \\
\qquad e2 : o2.events;\ s1 : o1.states;\ s2 : o2.states \bullet \\
\qquad\qquad d.methods = \{m2\} \\
\qquad\qquad \land\ d.events = \{e1\} \\
\qquad\qquad \land\ ((s1, m1), (s1, e1)) \in o1.transitions \\
\qquad\qquad \land\ ((s2, m2), (s2, e2)) \in o2.transitions
\end{array}
$$

To see that sys_{dist} is a valid counter-example, we observe the following trace:

$$
\langle (\{m1 \mapsto 1\}, \varnothing), (\varnothing, \varnothing), (\varnothing, \varnothing), (\varnothing, \{e2 \mapsto 1\}) \rangle
$$

The first step is a transition of $o1$, the second of d, and the third of $o2$. Neither of the transitions after the first decreases the size of the bag of methods, and therefore cannot be duplicated by any *SingleObject*:

$$\neg \; \exists so : SingleObject \bullet (sys_{dist}, so) \in equiv_{ES}$$

which contradicts the encapsulation property.

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 for which the encapsulation property holds. 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 Conclusions

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 and incomplete specifications can be avoided without abandoning the architectural paradigm. Second, a formal understanding of classes of systems aids reasoning about properties of styles. Such "analytic leverage" is only partly a consequence of having formal machinery to push through proofs. Arguably much more important is the fact that the exercise of producing a formal model (a) helps suggest the right questions to ask (e.g., does the encapsulation property hold?), and (b) leads one to determine with some precision where the problems are likely to arise. In this respect the formalism acts as a tool that focuses and augments our intuition and informal reasoning skills, rather than supplanting them.

However, as with any proposed use of formalism in software engineering, it is important to look not only at the benefits, but also the costs. Indeed, most attempts to apply formal methods to real software systems have failed in large measure because the costs of producing a formal model are simply too high.

There are several reasons why the kinds of formalisms proposed in this paper stand better long-term prospects for success. First, rather than specificing individual systems, we are focusing on the problem of specifying *families* of systems. Thus the effort required to produce the specification can be amortized over a potentially large number of specific systems.

Second, the result of style analysis is a set of general theorems about all systems in the family. (For example, we showed that all legal pipe-filter systems can be decomposed hierarchically, and furthermore they can be implemented using finite buffers if they satisfy certain additional contraints.) Consequently, the implementor of a specific system can simply rely on these general results, without having to deal with the formal model itself.

Third, a key component of our approach is the use of a common framework for style specification. This means that a new style can often be specified as an incremental modification to an existing style (e.g., as we did for pipelines). Even when a completely new style must be defined (as with ES), the framework provides considerable reuse, since the syntactic basis remains the same, and the form of specification is narrowly prescribed.

Finally, the approach we have outlined above permits varying degrees of formality within the overall framework. It is possible to define a style at a much less detailed level than we did with the two main examples of this paper. Those wishing to use the structure to model only a few aspects of a style can do so, while others can use it to produce the more detailed models illustrated in this paper.

Acknowledgments

A 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 [14] and [2].

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

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: Towns, Buildings, Construction.* 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 Vallen, 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 A.I.Wasserman. *Tutorial on Software Design Techniques.* IEEE Computer Society Press, 1976.

[10] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Design.* Addison-Wesley, 1994.

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

[12] 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.

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

[14] 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.

[15] 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).

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

[17] 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, Special Issue on Software Architecture 21*, 4 (April 1995), 288–301.

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

[19] Inverardi, P., and Wolf, A. Formal specification and analysis of software architectures using the chemical, abstract machine model. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture 21*, 4 (April 1995), 373–386.

[20] 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, Special Issue on Software Architecture 21*, 4 (April 1995), 336–355.

[21] Magee, J., and Kramer, J. Modelling distributed software architectures. In *Proceedings of the First International Workshop on Architectures for Software Systems* (April 1995), Reissued as Carnegie Mellon University Technical Report CMU-CS-95-151.

[22] 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.

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

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

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

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

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

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

[29] 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.

[30] Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture 21*, 4 (April 1995), 314–335.

[31] Shaw, M., and Garlan, D. Formulations and formalisms in software architecture. vol. 1000 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

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

[33] 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.

[34] 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 [32].

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.
- $\operatorname{seq} X$ is the set of all sequences, or lists, of elements from X, including empty and infinite sequences.

- 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 $X \times Y$.
  - $X \mathrel{+\!\!\!\!\rightarrow} Y$ 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 $\operatorname{seq} X$ 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 $\operatorname{bag} X$ 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

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;\mathit{EssentialProceedings}\;}} \\
\mathit{Proceedings} \\
\hline
\mathit{authors} = \operatorname{dom} \mathit{index} \\
\hline
\end{array}
$$

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 $\Delta \mathit{Schema}$ 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

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;\mathit{ProceedingsOp}\;}} \\
\Delta \mathit{Proceedings} \\
\hline
\end{array}
$$

which is equivalent to

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;\mathit{ProceedingsOp}\;}} \\
\mathit{Proceedings} \\
\mathit{Proceedings}' \\
\hline
\end{array}
$$

or

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;\mathit{ProceedingsOp}\;}} \\
\mathit{authors} : \mathbb{P}\, \mathit{AUTHOR} \\
\mathit{index} : \mathit{AUTHOR} \leftrightarrow \mathit{PAPER} \\
\mathit{authors}' : \mathbb{P}\, \mathit{AUTHOR} \\
\mathit{index}' : \mathit{AUTHOR} \leftrightarrow \mathit{PAPER} \\
\hline
\end{array}
$$

Some other operations and notational conventions used in Z are:

- $\mathit{Point} == \mathbb{N} \times \mathbb{N}$ 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 $\operatorname{dom} f$ is the domain of f and $\operatorname{ran} f$ 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*.

B Proof Sketch for Finitely-Implementable Criterion

First, we provide a means of identifying what computation is represented by a step in a computation trace:

$$
\begin{array}{|l}
ftrace : \mathbb{P}(\mathbb{N} \times FilterStep \times \mathrm{seq}\, SystemState) ptrace : \mathbb{P}(\mathbb{N} \times PipeStep \times \mathrm{seq}\, SystemState) \\
\hline
ftrace = \{ i : \mathbb{N};\ fs : FilterStep;\ tr : \mathrm{seq}\, SystemState \mid \\
\quad fs \upharpoonright FilterState \in tr(i).filter\_states \land fs \upharpoonright FilterState' \in tr(i+1).filter\_states \} \\
ptrace = \{ i : \mathbb{N};\ ps : PipeStep;\ tr : \mathrm{seq}\, SystemState \mid \\
\quad ps \upharpoonright PipeState \in tr(i).pipe\_states \land ps \upharpoonright PipeState' \in tr(i+1).pipe\_states \}
\end{array}
$$

Given these definitions, we must show that it is possible to reorder the computation of a pipe-filter system, under certain circumstances. Here is an example, which indicates that independent filter computations can be swapped:

$$
\begin{array}{l}
\forall\, fs_1, fs_2 : FilterStep;\ tr : \mathrm{seq}\, SystemState \\
\quad \mid tr \in LegalTracesOf(tr(1).sys) \land fs_1.filter \neq fs_2.filter \\
\qquad \land ftrace(\#tr - 2, fs_1, tr) \land ftrace(\#tr - 1, fs_2, tr) \\
\quad \bullet \exists\, tr_2 : LegalTracesOf(tr(1).sys) \mid \\
\qquad \#tr = \#tr_2 \\
\qquad \land (1 \mathinner{..} \#tr - 2 \lhd tr) = (1 \mathinner{..} \#tr - 2 \lhd tr_2) \\
\qquad \land ftrace(\#tr - 2, fs_2, tr_2) \land ftrace(\#tr - 1, fs_1, tr_2)
\end{array}
$$

This can be proved using the definition *SystemFilterStep*: Because the sequences tr and tr_2 are identical up to the last two steps, the state $tr(\#tr - 2)$ is identical to the state $tr_2(\#tr - 2)$. By the definition of *ftrace*, we know that $fs_2 \upharpoonright FilterState \in tr(\#tr - 1).filter\_states$. Because we know (from *LegalTracesOf* and $ftrace(\#tr - 2, fs_1, tr)$) that $tr(\#tr - 2)$ and $tr(\#tr - 1)$ are related by *SystemFilterStep*, and because (by assumption) fs_1 and fs_2 do not refer to the same filter, this implies that $fs_2 \upharpoonright FilterState \in tr(\#tr - 1)$. By definition of *LegalTracesOf* and *SystemFilterStep*, this implies that we can extend $1 \mathinner{..} \#tr - 2 \lhd tr$ so that $ftrace(\#tr - 2, fs_2, 1 \mathinner{..} \#tr - 1 \lhd tr_2)$. By a similar argument, we can show that $fs_1 \upharpoonright FilterState \in tr_2(\#tr - 1)$ and so tr_2 can be constructed such that $ftrace(\#tr - 1, fs_1, tr_2)$, which proves the property. This property and others like it allow us to reorder a given computation trace.

Next we must show that whenever a filter's buffers become full in a legal computation, there must be a later computation that will reduce their size:

$$
\begin{array}{l}
\forall\, i : \mathbb{N};\ tr : \mathrm{seq}\, SystemState;\ f : FilterState \\
\quad \mid tr \in TerminationsOf(tr(1).sys) \land f \in tr(i).filter\_states \\
\quad\ \land tr(1).sys \text{ is balanced } etc. \text{ with buffer size } n \\
\quad\ \land \exists\, p : f.filter.inputs \bullet \#f.instate(p) = n \\
\quad \bullet \exists\, j : i + 1 \mathinner{..} \#tr;\ fs : FilterStep \\
\qquad \mid ftrace(j, f_2, tr) \land f_2.filter = f.filter
\end{array}
$$

With this result, we can complete the proof: Consider any trace of a system that violates the buffer size constraint. Take the first computation step that would cause a buffer size to exceed the limit. Because the buffer is already full, we know that there must be a computation later in the sequence that reads from that buffer. Because the system is acyclic, we know that the buffer cannot be both read and written by the same component, so it must be possible to move the computation that reads the buffer to before the computation that would write it. Once this move is made, the writing computation will no longer cause the buffer size limit to be exceeded. This reordering process can be continued with any later computations that would exceed the buffer size limit. Eventually all steps that violate the buffer size constraint will have been moved to a safe place in the computation, and the reordered trace be in the set $\mathit{FiniteTerminationsOf}(n, \mathit{system})$.

C Configuration Map for ES

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