Towards Formalized Software Architectures

Robert Allen David Garlan

July, 1992

CMU-CS-92-163

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This is an expanded version of "A Formal Approach to Software Architectures," Proc. IFIP Congress '92, September 1992.

Abstract

An important goal in software engineering is to describe complex software systems at an architectural level of abstraction. While there are many useful architectural paradigms (pipes, blackboards, etc.) they are typically understood only idiomatically and applied in an ad hoc fashion. We show how a formal model allows us to say precisely what we mean by a software architecture, explore its properties, and systematically describe instances of the architecture. We illustrate the approach using the well-known example of pipe-filter architectures.

This research was sponsored by the National Science Foundation under Grants CCR-9109469 and CCR-9112880, and by Siemens Corporate Research, Inc. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Siemens or the U.S. Government.

Keywords: Dataflow Systems, Pipes and Filters, Software Architecture, Software Engineering, Software Design, Formal Specification.

1 Introduction

An important problem in software engineering is to characterize complex systems at an architectural level of abstraction. At this level the gross structure of a system becomes apparent in terms of high-level components and connections between those components. For example, a designer might characterize the structure of a compiler in terms of a "pipeline architecture," or parts of an operating system in terms of a "layered architecture."

A significant goal for understanding and describing systems at this level of design is the development of a taxonomy of software architectures. A taxonomy of well-understood software architectures will give software developers a common vocabulary for designing software systems; it will define a space of engineering alternatives for architectural models; it will allow developers to engage in software system design on a level more closely related to the structure of the system than is possible with current programming languages; and it will support reuse of existing designs and implementations.

Currently, efforts to develop such a taxonomy fall into two categories. One is "overview" research, which analyzes the similarities and differences between architectures and attempts to map out the shape of the whole field. The aim of this work is to identify and codify commonly used architectural paradigms, and to understand the tradeoffs in choosing one paradigm over another [15]. At the other extreme is the investigation of specific software structures. These studies typically elucidate the structure of specific software systems by describing them as high-level components and connections. This work aims to document the structure of a single system or class of related systems, so that similar applications can reuse the previous design. Examples include the ISO OSI model [13] and the "Toaster Model" [6].

Both efforts rely on the existence of a body of well-understood architectural abstractions. Research into specific systems results in statements of the form, "This program is an example of architecture X." Overview research results in statements such as "X is an example of a software architecture," or "Architectures X and Y are related as follows."

However, such statements beg the essential questions: "What is architecture X?" and "How is architecture X defined?" The absence of precise answers to these questions makes it difficult to evaluate claims about an architecture or to compare architectures. Moreover, without precise characterizations of specific architectures, practitioners have little hope of knowing how to apply those architectures to designs.

In this paper we argue that formal definitions of software architectures can help solve these problems. Specifically, we show how a formal model can provide a precise definition of a software architecture, thereby clarifying and extending our intuitive understanding. We illustrate the approach in terms of a pipe-filter software architecture.

In section 2 we provide an informal introduction to the pipe-filter architecture, and in section 3 we briefly describe the formal notation used in the paper. Then in section 4 we present the formal definition of the architecture. In section 5, we explore the formal model in more detail, and discuss how it may be used. Finally in section 6 we outline ways in which future research can build on the model we have presented, and, more generally, on the use of formal models for software architectures.

2 An Informal Description of the Pipe-Filter Architecture

A common way of structuring a program that transforms data is to divide it into a collection of incremental transformations. The sub-transformations are then combined, the outputs of certain transformations serving as inputs to others. The entire application is then the composition of the transformations. Such systems are called coarse-grained dataflow or pipe-filter systems. Unix pipes are perhaps the most familiar example, although there are many others [1, 2].

The components of a pipe-filter architecture, *filters*, transform streams of data. Each filter has *input ports*, from which it reads data, and *output ports*, to which it writes results. A filter performs its computation incrementally and locally: a single computational step consists of reading a portion of the data available at its input ports, transforming that data, and writing it to the output ports.

The connectors of a pipe-filter architecture, *pipes*, control the flow of data through the system. Each pipe links an output port to an input port, indicating the path that data will take and carrying out the transmission. Thus, a computational step in a pipe-filter system consists of either of two things: an incremental transformation of data by a filter, or a communication of data between ports by a pipe. In the context of an entire system, filters and pipes operate concurrently.

While the general notion of a pipe-filter architecture is widespread and useful, such an informal description raises many questions. Can a port be connected to more than one other port? Can there exist cycles in a graph of connections? What constitutes a complete computation? As we will see, these sorts of questions are resolved by a formal definition.

3 The Z Specification Language

To describe the pipe-filter architecture formally we use the Z specification language [16]. In this section, we briefly introduce this notation.

In Z a *schema* defines an object type by naming a set of *attributes*. Each attribute is defined by a (set-theoretic) type expression. Such types are constructed using primitive types and various type constructors. A primitive type is introduced in brackets.

$[PRIMITIVE]$

The type constructors we use in this paper are as follows:

- $\mathbb{P}$ provides the power set of another set;
- $\nrightarrow$ indicates the set of partial functions from one set to another;
- $\leftrightarrow$ defines a relation over two sets;
- $\times$ is the cross product of two sets; and
- seq is a sequence of elements of a set.

A schema further defines a type by predicates written below a dividing line. These define an *invariant* which must be satisfied by any instance of the type defined by the schema. In this paper, we use the following operators in our expressions:

- (x, y): an element of a cross product (or of a relation);
- dom, ran: the domain and range of a function;
- #: the size of a set (or a sequence);
- $\frown$: the concatenation of two sequences;
- $\forall x \mid y \bullet z$: for all x such that y holds, z holds;
- $\exists x \mid y \bullet z$: there exists an x such that y and z hold;
- $\wedge, \vee, \Rightarrow, =, \neq$: logical operators;
- $\{variables \mid predicate \bullet expression\}$: the set of values of the expression when it ranges over those variables which meet the predicate (see the example below);

- $\subseteq, \cap, \cup, \in, \varnothing, \setminus$: set operators;
- $\leq, >$: arithmetic comparisons;

Thus in the schema *Example*, the attribute *simple* must be an element of the primitive type *PRIMITIVE*, *set* must be an element of the powerset of *PRIMITIVE* (or a set of *PRIMITIVE*), *set*2 is a relation over *PRIMITIVE*, and *fun* must be a partial function from *PRIMITIVE* to pairs of *PRIMITIVE*. The first predicate states that for any instance of type *Example*, its *fun* attribute must be defined over values of *set*. The second predicate defines *set*2 to be the set of pairs of *PRIMITIVE* in the range of *fun* such that the first element is in *set*.

Example

$$
\begin{array}{|l}
\mathit{simple} : \mathit{PRIMITIVE} \\
\mathit{set} : \mathbb{P}\, \mathit{PRIMITIVE} \\
\mathit{set}2 : \mathit{PRIMITIVE} \leftrightarrow \mathit{PRIMITIVE} \\
\mathit{fun} : \mathit{PRIMITIVE} \nrightarrow (\mathit{PRIMITIVE} \times \mathit{PRIMITIVE}) \\
\hline
\operatorname{dom} \mathit{fun} = \mathit{set} \\
\mathit{set}2 = \{p1 : \mathit{set};\ p2 : \mathit{PRIMITIVE} \mid (p1, p2) \in \operatorname{ran} \mathit{fun} \bullet (p1, p2)\}
\end{array}
$$

4 The Formal Model

To define the pipe-filter architecture formally we proceed in three steps. First, we define *filters*, the basic component of the architecture. Second, we define *pipes*, the basic connector of the architecture. Third, we show how pipes and filters are combined to form a complete system. For each of these three aspects of the architecture we characterize both its static and its dynamic properties. To do this we show how state is associated with each filter, pipe, and pipe-filter system. Then we show how a computation step can change that state.

4.1 Filters

A *Filter* is defined by its name, its ports, and its program. A filter's name is taken from a primitive set of names, *FILTER*.

[*FILTER*]

Similarly, the ports of a filter are taken from a set of names, *PORT*.

[*PORT*]

To model the idea that ports are directional, we partition the ports of a filter into a set of input ports and a set of output ports. We associate a type with each port. This represents the kind of data that the filter is prepared to process on that port. Note that this refines the informal model that filters and pipes operate over streams of data. In the formal model we see that these streams are actually *typed* data. In characterizing the types of data we adopt a simple model: each type is an *alphabet*, a subset of the uninterpreted set *DATA*.

$[DATA]$

The filter's program is consists of three parts:

1. a set of legal program states, taken from the set of all possible states;

   $[FSTATE]$

2. a start state; and

3. a mapping from inputs to outputs, with a change in state possible as a side effect. Because a filter operates incrementally, the input data that a transition considers is only part of the total data that may be available to the filter. Similarly, in a single step it generates only a part of the total output. For this reason, we introduce a type *Partial_Port_State*, which indicates a mapping from ports to *part* of the stream present at that port.

   $Partial\_Port\_State == PORT \pfun \operatorname{seq} DATA$

This provides a *state machine* view of a filter. The filter invariant includes predicates that act as consistency checks, ensuring that the filter program respects its port types and cannot put the filter in an illegal state.

$$
\begin{array}{|l}
\hline
\textit{Filter} \\
\hline
\mathit{filter\_id} : \mathit{FILTER} \\
\mathit{in\_ports}, \mathit{out\_ports} : \mathbb{P}\, \mathit{PORT} \\
\mathit{alphabets} : \mathit{PORT} \mathrel{+\!\!\!\!\!\rightarrow} \mathbb{P}\, \mathit{DATA} \\
\mathit{states} : \mathbb{P}\, \mathit{FSTATE} \\
\mathit{start} : \mathit{FSTATE} \\
\mathit{transitions} : (\mathit{FSTATE} \times (\mathit{Partial\_Port\_State})) \\
\qquad\qquad\quad \leftrightarrow (\mathit{FSTATE} \times (\mathit{Partial\_Port\_State})) \\
\hline
\mathit{start} \in \mathit{states} \\
\mathit{in\_ports} \cap \mathit{out\_ports} = \varnothing \\
\operatorname{dom} \mathit{alphabets} = \mathit{in\_ports} \cup \mathit{out\_ports} \\
((s_1, \mathit{input\_observed}), (s_2, \mathit{output\_generated})) \in \mathit{transitions} \Rightarrow \\
\qquad s_1 \in \mathit{states} \land s_2 \in \mathit{states} \\
\qquad \land \operatorname{dom} \mathit{input\_observed} = \mathit{in\_ports} \\
\qquad \land \operatorname{dom} \mathit{output\_generated} = \mathit{out\_ports} \\
\qquad \land (\forall p : \mathit{in\_ports} \bullet \operatorname{ran}(\mathit{input\_observed}(p)) \subseteq \mathit{alphabets}(p)) \\
\qquad \land (\forall p : \mathit{out\_ports} \bullet \operatorname{ran}(\mathit{output\_generated}(p)) \subseteq \mathit{alphabets}(p)) \\
\hline
\end{array}
$$

The state of a filter during a computation (*Filter_State*) is composed of the current program state (*internal_state*), the data that is pending at an input port (*input_state*), and the data that has been written to an output port (*output_state*). The type of *input_state* and *output_state* is a mapping from ports to a stream of data. These represent *all* the data that is available.

$$\mathit{Port\_State} == \mathit{PORT} \mathrel{+\!\!\!\!\!\rightarrow} \operatorname{seq} \mathit{DATA}$$

$$
\begin{array}{|l}
\hline
\textit{Filter\_State} \\
\hline
f : \mathit{Filter} \\
\mathit{internal\_state} : \mathit{FSTATE} \\
\mathit{input\_state}, \mathit{output\_state} : \mathit{Port\_State} \\
\hline
\mathit{internal\_state} \in f.\mathit{states} \\
\operatorname{dom} \mathit{input\_state} = f.\mathit{in\_ports} \\
\operatorname{dom} \mathit{output\_state} = f.\mathit{out\_ports} \\
\forall p : f.\mathit{in\_ports} \bullet \operatorname{ran}(\mathit{input\_state}(p)) \subseteq f.\mathit{alphabets}(p) \\
\forall p : f.\mathit{out\_ports} \bullet \operatorname{ran}(\mathit{output\_state}(p)) \subseteq f.\mathit{alphabets}(p) \\
\hline
\end{array}
$$

A computational step of a filter consists of reading some of the data from the inputs and writing data to the outputs. The relation between the two is based on those inputs, the internal state, and the program. In Z, we describe such a transformation on state using a schema that contains two copies of the state (indicated by $\Delta Filter\_State$), together with a predicate that relates those copies. Unmarked attributes, such as *internal_state*, refer to the attribute values before the operation, while primed attributes, such as $internal\_state'$, refer to the values after the operation.

$$
\begin{array}{|l}
\hline
\textit{Filter\_Compute} \\
\Delta Filter\_State \\
\hline
f = f' \\
\exists\, input\_consumed, output\_generated : Partial\_Port\_State \bullet \\
\quad ((internal\_state, input\_consumed), \\
\quad (internal\_state', output\_generated)) \in f.transitions \\
\quad \wedge\ (\forall\, p : f.in\_ports \bullet \\
\qquad input\_state(p) = input\_consumed(p) \frown input\_state'(p)) \\
\quad \wedge\ (\forall\, p : f.out\_ports \bullet \\
\qquad output\_state(p) \frown output\_generated(p) = output\_state'(p)) \\
\hline
\end{array}
$$

These definitions formalize the intuition that a filter operates by transforming data incrementally. We see that the program defining the filter, *transitions*, is a local mapping from input data to output data: the computation and the output generated can only depend on what is actually consumed, not on data yet to be consumed. This schema also clarifies the idea of "local" transformation: a filter *is* allowed to depend on computations it has performed in the past (*FSTATE* appears in the *transitions* relation), but it is not allowed to depend directly on anything outside the filter or on what it has written out (the "preconditions" of *transitions* are limited to *in_ports*). This specifically precludes, for example, any data sharing between filters except that which is explicitly output on ports.

4.2 Pipes

A connector in a software architecture fills three roles:

1. It defines a protocol between components — the pattern and semantics of communication.

2. It defines a communication topology — which components communicate with which other components.

3. It defines a concrete execution model for carrying out that communication.

For pipe-filter systems there is a single type of connector, namely a pipe. Statically, a pipe is a typed connection between two ports, one an output of a filter and the other an input to a filter. Thus a pipe partially defines the topology of the system by specifying a pair of ports to connect.

$$
\begin{array}{|l}
\hline
\textit{Pipe} \\
\textit{source\_filter}, \textit{sink\_filter} : \textit{Filter} \\
\textit{source\_port}, \textit{sink\_port} : \textit{PORT} \\
\textit{alphabet} : \mathbb{P}\ \textit{DATA} \\
\hline
\textit{source\_port} \in \textit{source\_filter}.\textit{out\_ports} \\
\textit{sink\_port} \in \textit{sink\_filter}.\textit{in\_ports} \\
\textit{source\_filter}.\textit{alphabets}(\textit{source\_port}) = \textit{alphabet} \\
\textit{sink\_filter}.\textit{alphabets}(\textit{sink\_port}) = \textit{alphabet} \\
\hline
\end{array}
$$

The state of a pipe is divided into two parts: data that has been delivered to the sink and data that has not yet been delivered. We distinguish between these two sequences in order to represent the transmission of data across the pipe. This allows us to model the mechanism of communication explicitly but abstractly. (We discuss this further in section 5.2.)

$$
\begin{array}{|l}
\hline
\textit{Pipe\_State} \\
p : \textit{Pipe} \\
\textit{source\_data} : \mathrm{seq}\ \textit{DATA} \\
\textit{sink\_data} : \mathrm{seq}\ \textit{DATA} \\
\hline
\mathrm{ran}\ \textit{source\_data} \subseteq p.\textit{alphabet} \\
\mathrm{ran}\ \textit{sink\_data} \subseteq p.\textit{alphabet} \\
\hline
\end{array}
$$

An important property of this model is that pipes are self-contained entities and can be reasoned about independently of filters. A consequence of this approach is that the same data is characterized in two places in a specification: at the ports of the filters and at the ends of the pipes. Although an implementation would probably be forced to treat the data either as part of the filters or as part of the pipes, in the mathematical model such redundancy is not a problem: When we combine pipes and filters into a unified system, we will simply identify the state at each end of the pipe with the state at the port of the appropriate filter[1].

[1] This kind of identification is common in mathematical models of systems. For example in CSP two processes communicate by sharing a single event [10].

A computational step in a pipe delivers some of the data from its source to its sink. This definition highlights several aspects of pipes that were not clear in the informal definition. First, data is not altered during transmission. Second, the order of the transmitted data is not changed. Third, a pipe connects exactly two ports. However, the amount of data transmitted is not specified. Here again, the model is intentionally abstract to allow different implementations to adopt different data transmission policies.

$$
\begin{array}{|l}
\textit{Pipe\_Compute} \\
\hline
\Delta \textit{Pipe\_State} \\
\hline
p = p' \\
\exists\, \textit{deliver} : \operatorname{seq} \textit{DATA} \mid \#\textit{deliver} > 0 \bullet \\
\qquad \textit{source\_data} = \textit{deliver} \frown \textit{source\_data}' \\
\qquad \wedge\ \textit{sink\_data}' = \textit{sink\_data} \frown \textit{deliver} \\
\hline
\end{array}
$$

4.3 Pipe-Filter System

Next we describe how these objects are composed into a complete pipe-filter system. A system is defined as a collection of filters and a collection of pipes.

$$
\begin{array}{|l}
\textit{System} \\
\hline
\textit{filters} : \mathbb{P}\, \textit{Filter} \\
\textit{pipes} : \mathbb{P}\, \textit{Pipe} \\
\hline
\forall f_1, f_2 : \textit{filters} \bullet f_1.\textit{filter\_id} = f_2.\textit{filter\_id} \Leftrightarrow f_1 = f_2 \\
\forall p : \textit{pipes} \bullet p.\textit{source\_filter} \in \textit{filters} \wedge p.\textit{sink\_filter} \in \textit{filters} \\
\forall f : \textit{filters};\ pt : \textit{PORT} \mid pt \in f.\textit{in\_ports} \bullet \\
\qquad \#\{p : \textit{pipes} \mid f = p.\textit{sink\_filter} \wedge pt = p.\textit{sink\_port}\} \leq 1 \\
\forall f : \textit{filters};\ pt : \textit{PORT} \mid pt \in f.\textit{out\_ports} \bullet \\
\qquad \#\{p : \textit{pipes} \mid f = p.\textit{source\_filter} \wedge pt = p.\textit{source\_port}\} \leq 1 \\
\hline
\end{array}
$$

The consistency of a pipe-filter system is guaranteed by a number of requirements. First, each filter has a unique name. Second, there are no "dangling" pipes; both ends of every pipe must connect to a filter inside the system. This is an obvious requirement for being able to define a pipe-filter system without reference to other systems. But it also clarifies the topological role of pipes; pipes create the context within which filters operate, they do not operate independently of filters. Third, we require that ports connect to no more than one pipe. This serves to simplify the semantics of pipes and therefore of the architecture as a whole. If more complicated forms of data transmission are required — such as data broadcast — these must be modeled explicitly, by the use of an intermediate filter (see section 5.1) .

The model does not, however, require that every port of a filter be connected to *some* pipe. This permits us to describe open pipe-filter systems by treating unbound input or output ports as system inputs or outputs. Further, this allows us to characterize systems that can later be composed with other pipe-filter systems by adding new pipe connections. Finally, as we discuss later, this approach allows us to treat systems hierarchically by encapsulating any pipe-filter system as an equivalent higher-level filter.

Just as a pipe-filter system is defined by its parts, the state of a system is defined by the states of its parts. States are defined for exactly those filters and pipes in the system. It is at this point that we identify the states of ports and pipes.

System_State

$sys : System$
$filter\_states : \mathbb{P}\ Filter\_State$
$pipe\_states : \mathbb{P}\ Pipe\_State$

$sys.filters = \{fs : filter\_states \bullet fs.f\}$
$\forall fs_1, fs_2 : filter\_states \bullet fs_1.f = fs_2.f \Leftrightarrow fs_1 = fs_2$

$sys.pipes = \{ps : Pipe\_State \bullet ps.p\}$
$\forall ps_1, ps_2 : pipe\_states \bullet ps_1.p = ps_2.p \Leftrightarrow ps_1 = ps_2$

$\forall ps : pipe\_states \bullet \exists fs : filter\_states \bullet$
$\quad ps.p.source\_filter = fs.f$
$\quad \wedge\ ps.source\_data = fs.output\_state(ps.p.source\_port)$
$\forall ps : pipe\_states \bullet \exists fs : filter\_states \bullet$
$\quad ps.p.sink\_filter = fs.f$
$\quad \wedge\ ps.sink\_data = fs.input\_state(ps.p.sink\_port)$

A computational step in a pipe-filter system is either a computation step of a filter or a transmission of a pipe. The former is a straightforward extension of *Filter_Compute*: the system nondeterministically executes a single filter, leaving the rest of the system unchanged. The expression $\theta Filter\_State$ refers to the state of the filter before the computation, and $\theta Filter\_State'$ to the state after the operation.

$$
\begin{array}{l}
\textit{System\_Filter\_Step} \\
\hline
\Delta \textit{System\_State} \\
\hline
\textit{sys} = \textit{sys}' \\
\exists\, \textit{Filter\_Compute} \bullet \\
\quad \textit{filter\_states} \setminus \{\theta \textit{Filter\_State}\} = \textit{filter\_states}' \setminus \{\theta \textit{Filter\_State}'\} \\
\quad \wedge\ \theta \textit{Filter\_State} \in \textit{filter\_states} \\
\quad \wedge\ \theta \textit{Filter\_State}' \in \textit{filter\_states}'
\end{array}
$$

The definition of *System_Pipe_Step* is handled similarly. However, in this case it is somewhat more complicated to state that only the executing pipe changes. Because there is considerable state which exists in filters independent of pipes, we must explicitly state that it does not change. In particular, no filter's internal state, and no input or output ports except those of the specific pipe may change.

$$
\begin{array}{l}
\textit{System\_Pipe\_Step} \\
\hline
\Delta \textit{System\_State} \\
\hline
\textit{sys} = \textit{sys}' \\
\exists\, \textit{Pipe\_Compute} \bullet \\
\quad \theta \textit{Pipe\_State} \in \textit{pipe\_states} \\
\quad \wedge\ \theta \textit{Pipe\_State}' \in \textit{pipe\_states}' \\
\quad \wedge\ (\forall\, \textit{fil\_state} : \textit{filter\_states};\ \textit{fil\_state}' : \textit{filter\_states}' \\
\qquad \mid \textit{fil\_state}.f = \textit{fil\_state}'.f \\
\qquad \bullet\ \textit{fil\_state}.\textit{internal\_state} = \textit{fil\_state}'.\textit{internal\_state}) \\
\quad \wedge\ (\forall\, \textit{fil\_state} : \textit{filter\_states};\ \textit{fil\_state}' : \textit{filter\_states}'; \\
\qquad \textit{port} : \textit{PORT} \\
\qquad \mid \textit{fil\_state}.f = \textit{fil\_state}'.f \\
\qquad \wedge\ \textit{port} \in \textit{fil\_state}.f.\textit{in\_ports} \\
\qquad \wedge\ (p.\textit{sink\_filter} \neq \textit{fil\_state}.f \vee p.\textit{sink\_port} \neq \textit{port}) \\
\qquad \bullet\ \textit{fil\_state}.\textit{input\_state}(\textit{port}) = \textit{fil\_state}'.\textit{input\_state}(\textit{port})) \\
\quad \wedge\ (\forall\, \textit{fil\_state} : \textit{filter\_states};\ \textit{fil\_state}' : \textit{filter\_states}';\ \textit{port} : \textit{PORT} \\
\qquad \mid \textit{fil\_state}.f = \textit{fil\_state}'.f \\
\qquad \wedge\ \textit{port} \in \textit{fil\_state}.f.\textit{out\_ports} \\
\qquad \wedge\ (p.\textit{source\_filter} \neq \textit{fil\_state}.f \vee p.\textit{source\_port} \neq \textit{port}) \\
\qquad \bullet\ \textit{fil\_state}.\textit{output\_state}(\textit{port}) = \textit{fil\_state}'.\textit{output\_state}(\textit{port}))
\end{array}
$$

Now that we have defined *System_Filter_Step* and *System_Pipe_Step*, we can formally specify that these are the only legal system computation steps.

$$\textit{System\_Compute\_Step} \mathrel{\widehat{=}} \textit{System\_Filter\_Step} \vee \textit{System\_Pipe\_Step}$$

We now define a pipe-filter system computation as a sequence of steps beginning with a start state and progressing through legal system computation steps. The start state of a system is one in which every filter is in its *start* state, every pipe empty, and every output port contains no data. We do not require unconnected ports to be empty since, as explained earlier, these ports are treated as system inputs and outputs.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SystemStart}} \\
\hline
\textit{System\_State} \\
\hline
\forall \textit{fil\_state} : \textit{filter\_states};\ \textit{port} : \textit{PORT} \mid \textit{port} \in \textit{fil\_state}.f.\textit{out\_ports} \\
\quad \bullet\ \textit{fil\_state}.\textit{internal\_state} = \textit{fil\_state}.f.\textit{start} \\
\quad \wedge\ \#(\textit{fil\_state}.\textit{output\_state}(\textit{port})) = 0 \\
\forall \textit{pipe\_state} : \textit{pipe\_states} \\
\quad \bullet\ \#\textit{pipe\_state}.\textit{sink\_data} = \#\textit{pipe\_state}.\textit{source\_data} = 0 \\
\hline
\end{array}
$$

It is possible that a system may not terminate, given an initial start state. A system may terminate either because it has successfully processed all its data, or because there is no legal transition for any filter even though there is data at the ports. In either case, no further computations are possible. We express this in the schema *SystemFinal*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SystemFinal}} \\
\hline
\textit{System\_State} \\
\hline
\neg\ (\exists\, \textit{System\_State}' \bullet \textit{System\_Compute\_Step}) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CompleteComputation}} \\
\hline
\textit{trace} : \operatorname{seq} \textit{System\_State} \\
\hline
\exists\, \textit{sys} : \textit{System} \bullet \forall\, i : \operatorname{dom} \textit{trace} \bullet (\textit{trace}(i)).\textit{sys} = \textit{sys} \\
\exists\, \textit{SystemStart} \bullet \theta \textit{System\_State} = \textit{trace}(1) \\
\exists\, \textit{SystemFinal} \bullet \theta \textit{System\_State} = \textit{trace}(\#\textit{trace}) \\
\forall\, i : 1 \,..\, (\#\textit{trace} - 1) \\
\quad \bullet\ (\exists\, \textit{System\_State};\ \textit{System\_State}' \\
\qquad \bullet\ \theta \textit{System\_State} = \textit{trace}(i) \\
\qquad \wedge\ \theta \textit{System\_State}' = \textit{trace}(i+1) \\
\qquad \wedge\ \textit{System\_Compute\_Step}) \\
\hline
\end{array}
$$

Because this model does not specify the order in which the computations will be carried out it provides a concurrent model of system computation. That is to say, it models the fact that computations can occur in parallel by permitting them to complete in any order.

Two issues associated with concurrency that the model does not explicitly address are deadlock and fairness. Here, deadlock is not an issue, because there are no resources for which contention could be a problem. (Recall that the model allows pipes to buffer arbitrary amounts of data.) Because we only consider finite, complete computation sequences in this model, fairness is not an issue either; every filter or pipe which may execute must eventually execute, or the system cannot reach a final state. If we were to extend our model to include infinite computations, it would become necessary to state that the system is fair. This would, however, be a straightforward addition to our model.

5 Using the Model

A formal model such as this one provides a precise, mathematically-based description of a family of systems. The model attempts to capture a class of systems that is otherwise understood only idiomatically, and to expose the essential characteristics of that family while hiding unnecessary details. The value of this enterprise it that it allows us to pose and then answer fundamental questions about a software architecture. We now use the pipe-filter formal model to illustrate this point by considering three questions: (1) Does the model describe the right class of systems? (2) Is the model at an appropriate level of abstraction? (3) Does the model provide a natural way to describe and understand specializations of the architectural model?

The first of these questions allows us to understand whether we have captured the systems that correspond to our intuitive understanding of what the architecture represents. The second question allows us to understand whether the model includes sufficient detail to reason about important properties of those systems. The third question allows us to understand how to use the model to talk about specific systems within the broader space.

5.1 Class of Systems

A formal model defines a family of systems, which (under suitable abstraction) satisfy the mathematical properties of the model. Delimiting the "right" class of systems is a question of balance. The model should not exclude systems that are true instances of a pipe-filter architecture. But it *should* exclude other systems. To see how the pipe-filter model balances these requirements, consider design choices along two dimensions: computation and topology.

Computational Power: Computations in the pipe-filter model are determined by the computational power of both filters and pipes. Since a filter's computation is determined by a simple transition relation, it might seem that filters are too limited in computational power. This is not the case, however, since filters are Turing equivalent. To prove this, we let the internal filter states represent all possible combinations of Turing machine state, tape contents, and tape-head position. A Turing machine simulation then consists of reading in the complete input tape (marked by an end of string), simulating the Turing machine computation internally (using a potentially infinite *transitions* relation), and then, when the simulation halts, writing the output tape to an output port. If the Turing machine halts, then the filter writes out the correct output; if it fails to halt, then the filter will also fail to halt.

Given that a filter's computational model is not too restrictive, is it too general? An obvious restriction is to model filters as *finite* state machines. However, this leads to some rather unfortunate consequences. An important property of a pipe-filter architecture is that it is hierarchical: a pipe-filter system can be viewed, recursively, as a filter in another system. If the computational power of a filter is limited to a finite state machine, however, this property no longer holds. Since a pipe can hold an arbitrary amount of data, a system can assume an infinite number of states; there is no way to map an arbitrary system to a finite state filter.

In our model computations of pipes simply transmit data between ports of filters. One alternative is to relax this restriction and allow pipes to transform their data (as in [5]). However, this does not add power to the system: we can model a "computational" pipe as a pipe-filter-pipe sandwich in which the interposed filter performs the computation. But a more significant argument in favor of keeping pipe computations simple is that it better supports the philosophy of the architecture: the overall input-output behavior of a pipe-filter application can be understood as the simple composition of the behaviors of its filters. When pipes contribute to that behavior, much of the simplicity that motivated the architecture is lost. This simplicity translates formally to a straightforward mapping from this model into an abstract functional model.

Topological Constraints: The pipe-filter model prescribes a topology in which each pipe connects a single output filter port to a single input port. One alternative would be to allow a port to be connected to more than one other port. We chose not to do this because such an extension does not broaden the class of systems that can be modeled. For example, a one-to-many pipe can be obtained by interposing a "fan out" filter, having a single input and multiple output ports. Moreover, it is not clear what meaning to give to such topologies, since at least two possibilities are reasonable: data could be broadcast to all filters, or data could be sent to exactly one of the filters (as with Unix pipes). It would be unfortunate to have to pick one to the exclusion of the other, since clearly both policies make sense.

A second alternative would be to restrict the topologies that are allowed (for example, by excluding cycles). However, this kind of restriction would exclude from consideration the many useful pipe-filter systems that *do* allow arbitrary graphs. Moreover it is relatively easy to describe such topological limitations in the context of the more general model.

Consider, for example, a *pipeline*, a pipe-filter system where the filters are connected in a line. This can be modeled by adding a simple constraint to the basic architecture.

Pipeline

System

$$
\begin{array}{l}
\exists\, in, out : PORT;\ \ order : 1 \mathrel{..} \#filters \rightarrowtail filters \\
\qquad \bullet\ \#pipes = \#filters - 1 \\
\qquad \wedge\ (\forall f : filters \bullet f.in\_ports = \{in\} \wedge f.out\_ports = \{out\}) \\
\qquad \wedge\ (\forall p : pipes \bullet (\exists\, i : 1 \mathrel{..} \#filters - 1 \\
\qquad\qquad \bullet\ p.source\_port = out \wedge p.sink\_port = in \\
\qquad\qquad \wedge\ p.source\_filter = order(i) \wedge p.sink\_filter = order(i+1)))
\end{array}
$$

5.2 Level of Abstraction

In addition to describing a family of systems, a formal model also elucidates differences between systems within that family. The degree to which such systems can be distinguished is determined by the level of abstraction of the model. Again, finding the right level of abstraction is a question of balance: more abstract models may be simpler, but may not allow one to reason about important distinguishing properties.

An important aspect of the level of abstraction is the treatment of an architecture's operational semantics. One alternative to our incremental model would be to view filters as functions on streams, and to view pipes as defining a composition of these functions [11]. This is a higher level of abstraction, because it suppresses details of intermediate filter and pipe computations. This level of abstraction has the advantages that it more directly expresses the compositional nature of pipes and filters, and it makes it easier to reason about equivalence of filters. (See [4] for a formal example of this kind of architecture.) However, a functional model has several disadvantages relative to our model. Most importantly, it ignores aspects of the architecture that we have modeled, such as the incremental nature of filters and concurrency. These are important because (a) they model our intuition about the behavior of real pipe-filter systems, and (b) they allow us to make distinctions based on such things as the grain size of filter computation or scheduling discipline.

Alternatively, we could continue to model filters as we have, but not explicitly model the computation of pipes. Instead, connected filters would be viewed as sharing a queue between them; no pipe computation is necessary in this case because a filter computation would make data immediately available at the other end of the pipe. This is an attractive alternative since it is closer to the functional abstraction while still including much of the dynamic behavior of a pipe-filter system. Our model makes it easier, however, to define refinements of the architecture that are oriented towards implementation. These refinements might highlight network related properties, or performance modeling. In these refinements, the mechanism by which data is communicated from one filter to another is important. With a pipe computation explicitly described in the model, such mechanisms can be specified immediately; if this aspect of the architecture is abstracted away, then it is cumbersome to describe them without an intermediate step that essentially provides a model equivalent to ours.

Another approach would be to work at a lower level of abstraction. For example, one might introduce a more specific type system. However, this would limit the scope of the model to systems that use that type system. Moreover, the specific properties of the type system are less important than the fact that communicating filters agree on the kind of transmitted data. Adding a more concrete type system would complicate the model without clarifying properties specific to pipe-filter systems.

A second way to provide a more concrete model would be to constrain the use of non-determinism. One could, for example, introduce a particular scheduling discipline. This is unsatisfying since it would exclude systems that use alternative schedulers, and more importantly, as with CSP [10] and many other models, non-determinism is a natural way to model concurrency.

6 Future Work

The formal model presented in this paper illustrates how formalism can play a role in understanding a particular software architecture. As such, it represents a small step toward the much grander goal of developing a formal basis for software architecture. To achieve this goal the field will need to address several significant research issues.

First, and most obviously, is the need to develop formal models of more classes of systems. While some other systems have been treated in this way [9, 7, 8, 3, 11], few commonly used software architectures have been formalized.

Second is the need to develop more extensive models of specific architectures. In this paper we provided a static and dynamic model of pipes and filters, indicated how that model could be specialized, and discussed a number of its properties. However, there is considerable room for extension of these results. In particular, it should be possible to develop a complementary set of formal analysis techniques for pipes and filters. These would be expressed as a set of proof systems for reasoning about this class of system. For example, functional properties could be dealt with using a calculus of functional composition, while a different proof system would be needed for handling questions of performance. Others could be developed for reasoning about the correctness of a particular scheduler. What is significant about these proof systems is that they would be built upon the model of a specific architecture, rather than being general purpose proof systems.

Third is the need to compare alternative architectures with respect to their ability to address requirements of a software system. Currently software architectures are chosen on a largely ad hoc basis. The ability to formalize individual architectures suggests that it should be possible to use the formal properties in a practical way to make design choices [12]. In the same way that a good programmer might choose an algorithm or data structure based on its complexity analysis, we would hope that system designers could choose software architectures based on their formal properties. This should allow designers to make choices between different architectural paradigms as well as make selections among different specializations of the same architecture paradigm. This clearly requires analytical techniques that span individual architectures.

Fourth is the need to develop more appropriate notations for expressing and reasoning about software architecture. Our choice of Z in this paper was based on the ability of Z to support incremental development of specifications. On the other hand, Z has a number of limitations that make it awkward to describe architectural frameworks. In particular, it has weak support for parameterization. (This issue is discussed in more detail elsewhere [8].) It also provides little support for describing and reasoning about dynamic behavior and concurrency.

Fifth is the need to provide automated support for the description and use of architectural paradigms. Given a precise understanding of architectures and their properties, it should be possible to build specialized tools for creating, manipulating, reusing, specializing, and analyzing systems built using these architectures. Several tool development efforts are currently underway [14].

7 Conclusion

We have presented a formal model for the pipe-filter architecture in an attempt to define its essential characteristics and to expose the issues surrounding the definition of software architectures. It is this second point that is the more important—that modeling an architecture formally allows us to discuss its abstract properties with a precision unattainable in an informal setting.

There are many possible alternatives to the formal definition that we have presented here, some of which we considered in the previous section. It is quite possible that our model is not the best one, that some readers will strongly disagree with the characteristics of pipe-filter systems that we consider essential, and that others will argue that there are essential aspects of the architecture that we have omitted.

What is important about this model is that it precisely addresses the questions "What is the pipe-filter architecture?" and "How is the pipe-filter architecture defined?" As other models are proposed, and as other software architectures are modeled, a formal approach will allow much clearer discussions and explorations of architectures.

By requiring that statements about software architectures be made precise, the development of formal architectural definitions enriches and clarifies the value of software architecture. It acts both as a focus and as a starting point for the development of languages and tools for developing software systems, and it serves as a vehicle for communication about architectures.

Acknowledgements

We would like to thank Curtis Scott, Mary Shaw, and Pamela Zave for their helpful comments both on the formal model and on the description of it in earlier drafts of this paper. We would also like to thank Siemens Corporate Research, Inc. and the National Science Foundation for their support of this work.

A The Encapsulation Theorem

This section provides a formal statement of the theorem that a pipe-filter architecture is hierarchical – that is to say, a pipe-filter system can encapsulated as a filter.

The basic idea behind the theorem is that for any system there is a filter with equivalent computations. We will define a system's computations to be equivalent to a filter's if there is a correspondence between the ports of the filter and the unbound ports of the system such that for each step of the computation corresponding ports have the same state.

We begin by defining a legal trace to be one consisting of valid computation steps.

$$
\begin{array}{l}
\mathit{LegalTracesOf} : \mathit{System} \rightarrow \mathbb{P}(\mathrm{seq}\,\mathit{System\_State}) \\
\hline
\forall\, s : \mathit{System} \bullet \mathit{LegalTracesOf}(s) = \{\mathit{trace} : \mathrm{seq}\,\mathit{System\_State} \mid \\
\forall\, i : 1 \mathinner{..} (\#\mathit{trace} - 1) \\
\qquad \bullet\ (\exists\, \mathit{System\_State};\ \mathit{System\_State}' \\
\qquad\qquad \bullet\ \theta \mathit{System\_State} = \mathit{trace}(i) \\
\qquad\qquad \wedge\ \theta \mathit{System\_State}' = \mathit{trace}(i+1) \\
\qquad\qquad \wedge\ \mathit{System\_Compute\_Step})\}
\end{array}
$$

A *sysTrace* is a legal trace of any system.

$$
\begin{array}{l}
\mathit{sysTrace} : \mathbb{P}(\mathrm{seq}\,\mathit{System\_State}) \\
\hline
\mathit{sysTrace} = \{t : \mathrm{seq}\,\mathit{System\_State} \mid t \in \mathit{LegalTracesOf}((t(1)).\mathit{sys})\}
\end{array}
$$

We model a single filter as a system containing only one filter. This makes it easier to compare traces.

$$
\begin{array}{l}
\underline{\mathit{SingleFilter}} \\
\mathit{System} \\
\hline
\#\mathit{filters} = 1 \\
\#\mathit{pipes} = 0
\end{array}
$$

The unbound ports of a system are the ones that are not connected to any pipe in the system.

$$
\begin{array}{|l}
UnBoundPorts : System \nrightarrow \mathbb{P}(Filter \times PORT) \\
\hline
\forall s : System \bullet \\
UnBoundPorts(s) = \{f : s.filters;\ p : PORT \mid \\
\qquad p \in (f.in\_ports \cup f.out\_ports) \\
\qquad \land \neg\, (\exists\, pip : s.pipes \\
\qquad\qquad \bullet (f = pip.source\_filter \land p = pip.source\_port) \\
\qquad\qquad \lor (f = pip.sink\_filter \land p = pip.sink\_port)) \\
\bullet (f, p)\}
\end{array}
$$

A renaming between a system and a filter is a one-to-one correspondence between the unbound ports of the system and the ports of the filter.

$$
PortNameBinding == Filter \times PORT \rightarrowtail PORT
$$

$$
\begin{array}{|l}
Renaming : (System \times Filter) \rightarrow \mathbb{P}\ PortNameBinding \\
\hline
\forall s : System;\ f : Filter \bullet \\
Renaming(s, f) = \{r : PortNameBinding \mid \\
\qquad \operatorname{ran} r = (f.in\_ports \cup f.out\_ports) \\
\qquad \land \operatorname{dom} r = UnBoundPorts(s)\}
\end{array}
$$

Two computations are equivalent if for every step of each computation the states of the ports which correspond are equal.

$$
\begin{array}{|l}
EquivalentComps : \\
\qquad \mathbb{P}(sysTrace \times sysTrace \times PortNameBinding) \\
\hline
EquivalentComps = \\
\{t_1, t_2 : sysTrace;\ fil : Filter;\ rename : PortNameBinding \mid \\
\qquad (\exists\, SingleFilter \bullet (t_2(1)).sys = \theta SingleFilter) \\
\qquad \land \{fil\} = (t_2(1)).sys.filters \\
\qquad \land rename \in Renaming((t_1(1)).sys, fil) \\
\qquad \land \#t_1 = \#t_2 \\
\qquad \land (\forall\, i : 1 \mathinner{..} \#t_1;\ f : (t_1(1)).sys.filters;\ p : PORT \\
\qquad\qquad \mid (f, p) \in UnBoundPorts((t_1(1)).sys) \\
\qquad\qquad \bullet (\exists\, fs_1 : (t_1(i)).filter\_states;\ fs_2 : (t_2(i)).filter\_states \\
\qquad\qquad\qquad \bullet fs_1.f = f \\
\qquad\qquad\qquad \land p \in f.in\_ports \Rightarrow \\
\qquad\qquad\qquad\qquad fs_1.input\_state(p) = fs_2.input\_state(rename(f, p)) \\
\qquad\qquad\qquad \land p \in f.out\_ports \Rightarrow \\
\qquad\qquad\qquad\qquad fs_1.output\_state(p) = fs_2.output\_state(rename(f, p)))) \\
\bullet (t_1, t_2, rename)\}
\end{array}
$$

This theorem states that for every pipe-filter system there exists a single filter that performs equivalent computations. The function *rename* fills the role of mapping between inputs/outputs of the system to those of the filter.

Theorem: (Encapsulation)

$$
\begin{array}{l}
\forall\, s : \mathit{System} \bullet \exists\, f : \mathit{SingleFilter};\ \mathit{rename} : \mathit{PortNameBinding} \bullet \\
\quad (\forall\, T_1 : \mathit{LegalTracesOf}(s) \bullet (\exists\, T_2 : \mathit{LegalTracesOf}(f) \bullet \\
\qquad (T_1, T_2, \mathit{rename}) \in \mathit{EquivalentComps})) \\
\quad \wedge \\
\quad (\forall\, T_2 : \mathit{LegalTracesOf}(f) \bullet (\exists\, T_1 : \mathit{LegalTracesOf}(s) \bullet \\
\qquad (T_1, T_2, \mathit{rename}) \in \mathit{EquivalentComps}))
\end{array}
$$

References

[1] Barbacci, M. R., Weinstock, C. B., and Wing, J. M. *Programming at the Processor-Memory-Switch Level*. In **Proceedings of the Tenth International Conference on Software Engineering**. 1988.

[2] Browne, J. C., Azam, M., and Sobek, S. *Code: A Unified Approach to Parallel Programming*. **IEEE Software**, July 1989.

[3] Craig, I. D. **Formal Specification of Advanced AI Architectures**. Ellis Horwood Limited, Chichester, England, 1991.

[4] Delisle, N. and Garlan, D. *Applying Formal Specification to Industrial Problems: A Specification of an Oscilloscope.* **IEEE Software**, September 1990.

[5] Doubleday, D. L., Barbacci, M. R., and Weinstock, C. B. *Building Distributed Ada Applications from Specifications and Functional Components*. In **Proceedings of TRI-Ada '91**. 1991.

[6] Earl, A. *A Reference Model for Computer Assisted Software Engineering Environment Frameworks*. no. HPL-SEG-TN-90-11, Hewlett Packard Laboratories, Bristol, England, August 1990.

[7] Flinn, B. and Sorensen, I. H. *CAVIAR: A Case Study in Specification*. In **CAVIAR: A Case Study in Specification**, by B. Flinn and I. H. Sorensen, edited by I. Hayes. Prentice Hall International, 1987.

[8] Garlan, D. and Delisle, N. *Formal Specifications as Reusable Frameworks*. In **VDM '90: VDM and Z – Formal Methods in Software Development**. Springer-Verlag, LNCS 428, Kiel, Germany, 1990, pp. 150–163.

[9] Garlan, D. and Notkin, D. *Formalizing Design Spaces: Implicit Invocation Mechanisms*. In **VDM'91: Formal Software Development Methods**. Springer-Verlag, LNCS 551, 1991, pp. 31–44.

[10] Hoare, C. A. R. **Communicating Sequential Processes.** Prentice Hall, 1985.

[11] Kahn, G. *The Semantics of a Simple Language for Parallel Programming*. **Information Processing**, 1974.

[12] Lane, T. G. *User Interface Software Structures*. Carnegie Mellon University, School of Computer Science, 1990.

[13] McClain, G. R. **Open Systems Interconnection Handbook**. McGraw-Hill Book Company, New York, NY, 1990.

[14] **Proceedings of the Workshop on Domain-Specific Software Architectures**. Software Engineering Institute, 1990.

[15] Shaw, M. *Larger Scale Systems Require Higher Level Abstractions*. **Proceedings Fifth International Workshop on Software Specification and Design, IEEE Computer Society, Software Engineering Notes**, vol. 14 (1989), pp. 143–146.

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