The Aesop System: A Tutorial

Ralph Melton
The ABLE Project
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213

1 Introduction

The *Aesop* system is a family of environments for the design and analysis of software architectures. A *software architecture* is a description of a software system at a high level of abstraction, as a composition of computational elements and their interactions. Aesop is special because of its rich support for environments tailored to particular architectural styles. An *architectural style* is a recurring pattern of system organization. By exploiting information about architectural style, an Aesop design environment can guide software engineers in creating architectures and leverage in analyzing them.

Architectural designs are important for at least two reasons. In the first place, an architectural description provides an overview of a software system at a sufficiently abstract level to be comprehensible. In the second place, designing a system at an architectural level permits the reuse of structures at a high level of abstraction.

The use of architectural styles can have several benefits. It can promote design reuse by clarifying the context of applicability of particular solutions. Using architectural styles can also promote code reuse by permitting shared implementations of invariant aspects of an architectural style. Reuse of architectural styles can also make it easier for others to understand a system by fostering the use of conventionalized structures. Use of standardized styles also supports interoperability. An architectural style can also permit specialized style-specific analyses and even code generation. Furthermore, style-specific visualizations can be used to provide renderings that match the models used by domain experts.

This document is a guided and annotated tour of the Aesop system. It combines a script for walking through a demonstration of the system with an explanation of what the demonstration means and why the system acts in the ways that it does. Therefore, this document switches back and forth between two modes: in one mode, it discusses specific actions to do to follow along through the demo. (These actions are usually bracketed by lines, and headed with the words **To do:**) In the other mode, the document explains the meaning and significance of the actions of the program's behavior. This is not an exhaustive manual of all things that are possible with the Aesop system.

This document begins with the

2 Getting Started

To start Aesop, change to the `ui` directory, and type

```
% aesop
```

at the Unix system prompt. This will open the *Aesop Design Manager* window, with buttons for creating new designs, opening existing designs, copying designs, deleting designs, changing styles, or exiting Aesop.

To change the active style, click on the `Change Style` button in the Aesop *Design Manager* window, select a style from the list of styles, and click on the `OK` button. The name of the currently active style is displayed in the title bar of th*e Aesop Design Manager* window.

Four styles are included with this release of the Aesop system:

- The *Generic Style* provides only those generic elements of software architecture that are shared by all styles.
- The *Pipe and Filter Style* describes architectures that are composed of filters that transform streams of input data and pipes that provide sequential delivery of stream data between filters.
- The *Unix Pipe and Filter Style* extends the *Pipe and Filter Style* to include Unix binaries such as sort and grep, and support for files. With this style, one can produce designs that can automatically be implemented in the Unix environment.
- The *Real-Time Style* provides tools for analyzing real-time message-processing systems for schedulability and resource conflicts.

To open a design, click on the `Open` button in the *Fable Manager* window. Select a design from the scrolling list, and click on `OK`. Only designs created in the currently selected style will be visible in the list.

3 Aesop Basics: the Generic Style

The generic style presents only those elements of software architecture that are common to all styles: components, connectors, ports, roles, and configurations.

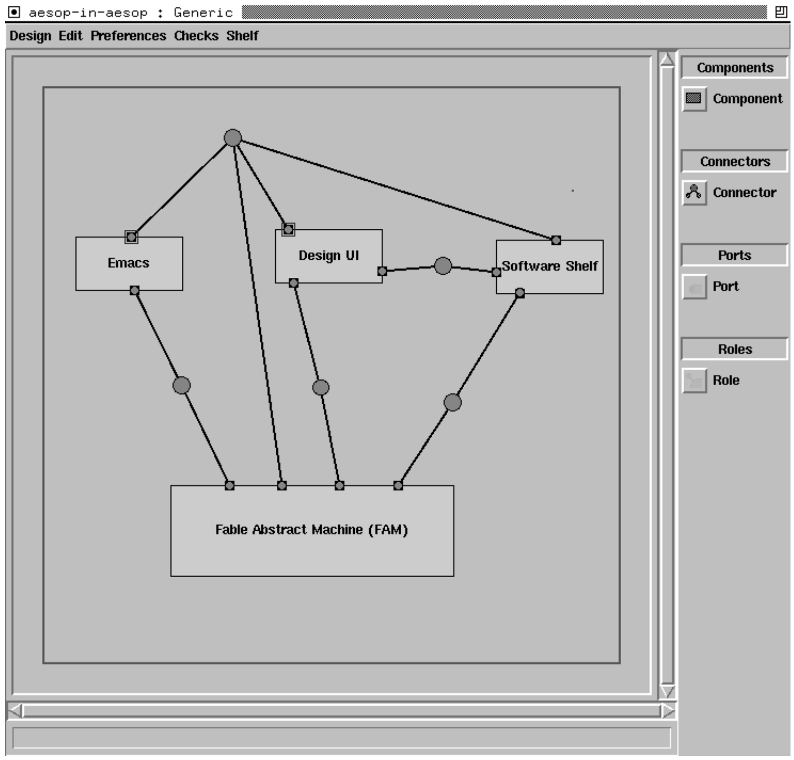
This section has two purposes. First, it explains the theory behind the designs that Aesop modifies. Second, it presents an overview of the design of the Aesop system itself by using the `aesop-in-aesop` design as its primary example.

To do:

- Check the title bar of the *Aesop Design Manager* window to confirm that you are in the generic style.
- To open a document: click on `Open` from the *Aesop Design Manager* window. Select a design from the scrolling window that appears, and click on the `OK` button.
- Open the `aesop-in-aesop` design, which presents the design of Aesop itself. This design will be used for examples in the rest of this section.

The design window has four major parts:

- An editable picture of the design itself takes up most of the window.
- A menu bar is at the top of the window.



- A palette of design elements that can be created and used in the current style, at the right side of the window.
- A status line at the bottom of the window that provides information about the element being pointed at by the mouse and messages from the results of user actions.

3.1 Design Elements: Components, Connectors, Ports, Roles, and Configurations.

The fundamental *design elements* of an architecture are components, connectors, ports, roles, and configurations. When the mouse pointer is over a design element, the name of the design element appears in the message line at the bottom of the window. Double-clicking on the design element brings up the element *workshop*, which presents information on a design element and allows that information to be modified.

Components are the application-level computations and data stores of a system. In the generic style, components are represented by rectangles. At the top level of the `aesop-in-aesop` design, there are four components, named `Emacs`, `Design UI`, `Software Shelf`, and `Fable Abstract Machine (FAM)`.

- The `Fable Abstract Machine (FAM)` stores designs in a database and provides an object-oriented interface to those designs.
- The `Design UI` provides a graphical interface to manipulate architectural designs. This is the tool that you are using right now. Part of the Aesop philosophy is that the user interface should be merely another tool for manipulating designs in the database.
- The `Software Shelf` facilitates reuse at an architectural level by storing individual components and connectors for later reuse.

- `Emacs` is an example of an external tool. Aesop strives to be an open environment, capable of very flexible integration with existing tools that were not written with Aesop in mind.

Components may have **ports**. A port defines a point of interaction between a component and its environment. In the generic style, ports are represented by dark squares on the edges of components. For example, the `Design UI` has three ports: a `FAM-RPC` port for the interaction with the `FAM-RPC` connector, an `Event Socket` port for the interaction with the event system, and a `Shared State` port for the interaction with the Software Shelf.

Connectors define the interactions between components. In the generic style, connectors are represented by lines with circles in the middle.The central knob of the connector is the place to double-click in order to view details about the connector. When the mouse pointer is over the central knob of a connector, the connector's name is shown in the message line at the bottom of the window. The top level of the `aesop-in-aesop` design uses three types of connectors:

- Three `Remote Procedure Call` connectors between tools and the FAM. Individual tools change the contents of the database by performing remote procedure calls.
- An `Event System` linking all three tools and the FAM. Tools are invoked and notified of changes to objects by broadcasting events. This connector illustrates that connectors are not necessarily binary.
- The `Software Shelf` and the `Design UI` use a different interaction than the event system allows, so there is `Shared State` between them.

Just as components have ports as their points of intersection with the rest of the system, connectors have **roles** as their access points. Roles are represented by small circles at the ends of the lines leading from the central knob of a connector.

A **configuration** is a composition of components and connectors into a system. Components and connectors are joined together by *attaching* ports and roles. Attaching a port to a role declares that the port will play that role in the interaction described by the connector. Various checks of compatibility between ports and roles are possible; for example, if both the port and the role have associated signatures, a type checker can provide one level of checks for compatibility. If both the port and the role have a description in the Wright language, a stronger check of compatibility is possible.

To do:

- Move the mouse over various components, connectors, ports, and roles. Note that in every case, the name of the element at which the mouse is pointing is displayed in the status bar at the bottom of the window.
- Click on various design elements to select them. Selected design elements are outlined in red.
- Create a new component by clicking on the `Component` button in the palette to the right of the window. When it is first created, it will be selected automatically.
- While that new component is selected, click on the `Port` button to create a new port on that component.
- Click on the `Connector` button to create a new connector. To change its location on the page, drag its center with the mouse.

- Select the new connector, and click on the `Role` button to create a new role.
- To attach the new role to the new port, drag the role onto the port, such that the hollow red square that you are dragging is around the black box of the port. To detach the role and the port, drag the role away from the port. When the mouse is over a role which is attached to a port, the status line shows '*Rolename* attached to *Portname*'

3.2 Workshops and Attributes

Sometimes, it is necessary to examine design elements in more detail than is provided by the high-level visual picture. Aesop provides *workshops* for each type of design element to provide access to more detail about an element. Workshops allow one to change the name of an element and its representations, and possibly other properties. Component workshops also provide access to the ports of the component, and connector workshops provide access to the connector's roles.

To do:

- Double-click on the `Design UI` component to open it in the *Component Workshop*.
- Use the editable text field to change the name of the component.
- (To confirm your changes and close the workshop, click on the `Close` button. Don't close the workshop yet, though; it will be used again in the next section.)

Every element also has a list of *attributes*. This is a list of name-value pairs. These attributes are not interpreted by the basic system; they are provided as an open-ended way of allowing for new types of data that may be used by humans or by other tools.

3.3 Representations and External Tools.

Any type of design element may have one or more *representations*. Representations may be files of source code, pieces of documentation, or decomposition of elements into configurations of their own. One of the major ways Aesop provides for integration with existing tools is by allowing for a wide variety of possible types of representations and mechanisms for examining them.

To do:

- With the `Design UI` still open in the *Component Workshop*, select the `Design UI Description` representation from the list of representations on the right.
- Click on the Get Info button to see the locations of files external to the database, the editors with which they are edited, and so forth.
- Click on the Edit button; The `Design UI description` will automatically be opened in your standard editor.

- Click on the `Close` button to dismiss the *Component Workshop*.

3.4 Hierarchical design in Aesop

The use of representations allows designs to be hierarchically decomposed by representing a component or connector as a subordinate design. When this is done, it is necessary to represent the fact that a port or role on the outer element corresponds to a port or role on the internal configuration. This correspondence is called a *binding*. Bindings differ from attachments in that a binding asserts that two ports or two roles in different configurations actually represent the same interaction, while an attachment joins a port and a role in a single configuration.

To do:

- Double-click on the `Fable Abstract Machine` component to bring up the *Component Workshop*.
- Select the `FAM Internals` representation, and click on the `Edit` button to open it.
- The representation of the component will be opened in a user-interface window of its own.
- In the window for the aggregate, the green lines connecting ports on the `Stylized Fcl Interpreter` to ports on the outer boundary of the design represent bindings. Bindings can be connected to ports and divorced from ports much the way roles can.
  Close the window from the Design menu when you are done.

3.5 First-class Connectors

In Aesop, connectors are truly first-class; they can have the same richness of detail and meaning that components do. In particular, connectors can be hierarchically decomposed in the same way components can be decomposed.[1]

To do:

- Double-click on the circle representing the event system.
- Select the `Event-System-Internals` representation, and click on the `Edit` button.
- This displays the architectural structure of the event system connector. The bindings work as described before.
- Close the window when you are done.

1. An argument for representation of connectors as first class semantic entities is beyond the scope of this paper, but can be found elsewhere [AG94a, Sha93].

- Choose *Close Design* from the *Design* menu to end this portion of the demo.

---

4 The Software Shelf: a Repository for Reuse

(This section may be skipped on first reading.)

Aesop tries to facilitate software reuse at the architectural level with a mechanism calledthe Software Shelf. This provides a database of architectural elements with a query engine for searching among elements and a mechanism for instantiating elements from the shelf when they are brought into designs.

To do:

- Choose *Open Shelf* from the *Shelf* menu.
- When the list of shelves appears, select `generic-shelf`, and click on `Set Target Repos`

The list box shows a list of design elements that can be incorporated into a design. The most common operation is to copy an element into a design.

To do:

- Select a design element in the scrolling list box.
- Choose *Transfer Current Selection* from the *Browser* menu.
- In the dialog that appears, edit values for the configuration options.
- Select a design to put the element in by clicking on the `Set Target Design` button.
- When everything is set to your liking, click on the Transfer button to copy the element into your design.

Of course, a repository would be useless without some way of adding design elements to it.

To do:

- To add an element from a design onto the shelf:
- Select the item in the design window.
- Choose *Put Selection on Shelf* from the *Shelf* menu.
- This form corresponds to the form that is invoked when you copy objects from the shelf; you can select a repository to transfer the selection to, and choose attribute values and configuration options to add to the object. Both of these lists must be lists of pairs, with each pair enclosed in curly braces. ‘{}’.
- When you have entered everything to your liking, click on the `Commit` button.

- To abort the transfer, click on the ‘Abort’ button instead.

---

5 Architectural Style: the Pipe and Filter Style

The generic style is applicable in a wide variety of contexts. However, it provides no semantics, and no guidance in design or analysis.

The key virtue of the Aesop system is that the generic design environment can be easily customized to support specialized styles of software architecture that do allow for guidance in design and analysis. One of the most well-understood styles is the *Pipe and Filter* style.

5.1 The Pipe and Filter Style

In the *Pipe and Filter* style, the primary components that are used are all filters, and the connectors that are used are all pipes. *Filters* transform input streams into output streams, with no shared state among different filters. *Pipes* provide sequential delivery of data streams between filters.

To do:

- Close any open designs.
- Change to the *Pipe and Filter* style.
- Open the `simple-pf-demo` system.

A filter may have two types of ports: *input* ports and *output* ports. These two ports are visually distinguished; input ports are drawn as a red triangle that points into the component, while output ports are drawn as a black triangle pointing out from the component. Similarly, pipes have two different types of roles: *source* roles, drawn in red, and *sink* roles, drawn in black.

The *Pipe and Filter* style enforces new constraints on how systems can be put together, also. In particular, it requires that source roles can only be attached to output ports, and sink roles can only be attached to input ports. (Visually, this means that a port and a role must be the same color to be attached.)

To do:

- Create a new filter by clicking on the `Filter` button. Notice that it is given an input port and an output port by default.
- Create a new pipe. Pipes likewise come with a source role and a sink role.
- Attach a source role (a red role) to an output port (a red port). If the port and role are not otherwise attached, the attachment should succeed.

- Try to attach a source (a red role) to an input port (a black port). It will fail.

By using the *Pipe and Filter* style, we have narrowed the space of possible designs we are considering. Within this style, you can only create designs that connect outputs of some filters to inputs of other filters via pipes that are connected properly. Within this space, we are able to give better feedback for flawed designs, by checking for problems such as ports and roles attached improperly. By using the *Pipe and Filter* style, we have gained leverage for designing architectures that are appropriate to that style.

6 A More Specific Substyle: Unix Pipes and Filters

The *Pipe and Filter* style can be refined even further to cover pipes and filters as they are used in the Unix operating system. Because the *Unix Pipe and Filter* style is based on the *Pipe and Filter* style, and because every design in the *Unix Pipe and Filter* style is a valid design in the *Pipe and Filter* style, we say that the *Unix Pipe and Filter* style is a *substyle* of the *Pipe and Filter* style. This substyle is understood thoroughly enough that we can generate code from the architectural description.

To do:

- Close all open designs.
- Change to the *Unix Pipe and Filter* style.
- Open the `simple-demo` system

In this style, the types of possible components are specialized even further into four categories:

- A `StdFilter` is a filter as in the *Pipe and Filter* style; no special support is provided for this type.
- A `UnixFilter` represents a normal Unix filter. It has ports `stdin`, `stdout`, and `stderr`, and its representations are by default edited with a style-specific external too, the *Filter Editor*.
- A `UnixBinary` *is* a filter corresponding to a standard unix binary, such as `sort` or `grep`.
- A `UnixFile` is a normal text file. Directing a pipe to or from a file has the same meaning as redirecting input to or from a file in most shells. Files may have at most one port.

Using a specialized style allows one to gain additional leverage through the use of style-specific external tools. For example, we have defined a simple language for writing filters which guarantees that every program written in this language will fulfill the requirements to be a filter.We built a syntax-directed structured editor for this language using a tool called the Synthesizer Generator.[2]

2. Add information about the Synthesizer Generator here.

To do:

- Double-click on the `Package` component to open the Filter Workshop.
- Select the `Package-fil-code` representation and click on the Edit button.
- The filter editor will be opened up on this representation.

This style is specialized enough to do code synthesis.

To do:

- To build an executable for this design, choose *Build* from the *PF-Tools* menu. The system will automatically compile the filter code into C, generate boiler-plate code to create processes and fork pipes, generate a Makefile for the system, and compile the whole system to produce an executable.

Some caveats are in order, because not all systems that can be created in the Unix-PF style are acceptable to the Build-PF tool.:

- For each filter *Filter* with a representation in the filter language, the representation must be in a file named *Filter*.fil, and that representation must define a procedure named *Filter*. (i.e., the component, the filename, and the procedure in the file must all have the same name.)
- The Build-PF tool does not generate a working program from a file connected by a pipe to another file. Similarly, a port on a file cannot be bound to an external port. The basic rule is that a file must always be connected to a filter.
- Build-PF does not seem to handle cycles of pipes and filters well. Some cycles of pipes and filters produce correct output, but do not terminate properly.
- If you do not have permissions to read the filter-language representations of a design, the Build-PF tool will fail. This is not a flaw in the Build-PF tool, but it is a common mistake.

7 Another Style: the Real-time Message-Passing Style

Most architectural descriptions will not be detailed enough to make code generation feasible. Architectural analysis can be useful, even so. The Real-time style demonstrates a style intended to provide leverage for analysis, instead of code-generation.

The Real-time style is based upon research done by Kevin Jeffay at the University of North Carolina. The Real-time style models systems that need to manipulate streams of data with tight scheduling concerns. Real-time processing of video data would fall into this category, for example. The environment for this style includes tools that analyze designs for schedulability on a uniprocessor. This hypothetical uniprocessor executes 512 ticks each second; each possible action requires an integral number of ticks.

To do:

- Close any open designs.
- Change to the *Real-time* style.
- Open the `msgdemo` system.

In this style, there are three types of components:

- `Devices` are sources of messages.Each device has three attributes: the type of message it carries, the rate at which it creates messages (in messages per second) and the time in clock ticks used for each message generated by the device. Devices can only have output ports.
- `Processes` are akin to filters in the pipe-and-filter style, for they transform streams of messages into streams. Each process has four distinguished attributes: the type of message of its input ports (which must all have the same type), a Transfer Function that declares how many messages it produces for each message it consumes, a cost in ticks for each message it produces, and a rate at which it must run to keep up with the data provided by its predecessors. (The rate of a process is a derived attribute; it can be calculated from the transfer functions of processes, the rates of devices, and the connections among processes and devices in the system.) Processes can have input ports, output ports, or synchronization ports.
- `Resources` represent system resources that are required for a process's execution, such as a hard disk or a floating point unit. Resources only have a cost in ticks per use. Resource components can only have synchronization ports.

An output port can also have a type of its own, representing the type of messages that are produced on that port.

There are two different types of connectors:

- `Asynchronous` connectors are akin to pipes; they transfer data asynchronously between data producers (devices and processes) and data consumers (processes). Asynchronous connectors have one derived attribute: the transfer rate of the connector in messages passing across the connector per second.

Asynchronous connectors may be clocked or unclocked. If a connector leading outward from a process is *clocked*, the process generates a message on that connector only when it has a message available on all of its input ports. If a connector leading outward from a process is *unclocked*, the process generates a message on the outbound connector when it receives a message on any of its input ports. Clocked connectors are represented by a stylized clock in the circle in the center of the connector.

Note: it might be simpler to have the clocked or unclocked nature be associated with the process, not the connector leading out from the process. We represent clockedness this way, because this is the way that is used by the domain experts with whom we developed this style. We consider it extremely important to support the styles of representation that are actually used by expert designers in their domains. Our environments should adapt to match the styles of designers and users, not the other way around.

- `Synchronous` connectors attach resources to processes that use those resources. They have no special attributes. Synchronous connectors must join a process and a resource.

Several types of checks and analyses are possible with the *Real-time* style. All these analyses are available from the *Tools* menu:

The *Reset Rates* command sets the rate for each process and asynchronous connector to 'undefined'. This is useful as a check that the Calculate Rates option is setting rates properly.

The *Set Processor Speed* command allows one to change the speed of the hypothetical uniprocessor upon which this system runs.

The *Check Connectors* command checks that, for each asynchronous connector, the types of the ports it joins have the same type. (Recall that the type of an input port is defined to be the type of its component.)

The *Check Processes* command checks that each process has at least one device feeding it.

The *Calculate Rates* command calculates the rate for each process and connector based on the device rates and transfer functions.

To do:

- Choose *Reset Rates* from the *Tools* menu.
- Double-click on a process to open the *Process Workshop*. Note that the rate of the process is set to 'undefined'.
- Click on `Accept` to close the *Process Workshop*.
- Choose *Calculate Rates* from the *Tools* menu.
- Double-click on the process again, and note that the rate of the process has been updated.

The *Schedulability Analysis* command combines two tests. It checks to see whether all the processes can be scheduled concurrently on a uniprocessor, and it checks to see whether there are any resource conflicts, such as contention for the hard disk.

To do:

- To create a processor overload, set the rate of one of the devices very high. (I usually set `Dev1`'s rate to 320.)
- Select *Do All Analyses* from the *Tools* menu.
- When the *Feasibility Workshop* window appears, click on the button that says `check to see if processor is overloaded`.
- It will display a window with a histogram of processor utilization and an analysis of how much processor time is required every second.
- Close these windows, and reset the rate of the device to remove the processor overload.

To do:

- To create a resource conflict, set the cost of one of the resources to a higher number. (I usually change the `FPU`'s cost from 4 to 8.)
- Select *Schedulability Analysis* from the *Tools* menu.
- When the *Feasibility Workshop* appears, click on the button that says `check for resource conflicts`.
- It will report a resource conflict.

Note that the feasibility analysis includes some guidance on how to fix resource conflicts. This analysis and guidance on correcting problems is a strong aid to design that is possible within a specialized architectural style.

8 Conclusions and Future Work

The Aesop system comes into its own not when it is considered as a particular design environment, but when it is considered as a family of related design environments.

We have several improvements to make in future work. We intend to simplify the process of defining new styles, so that slight modifications to existing styles require only slight changes. We intend to explore other architectural styles, such as layered systems, state machines, and client-server architectures. We intend to improve Aesop's support for designs that use multiple styles, which will require exploring what it means on a theoretical level for a design to partake of multiple styles. We intend to provide computer support for design patterns in Aesop. Aesop provides a rich basis for future research in computer support for architectural design.

We are always interested in user feedback and suggestions for improvements. Please send comments to `aesop-help@cs.cmu.edu`.

Appendix A. Maintenance

A.1 Restoring the Demo

To restore the demos that we ship to their original state, run the following shell script:at the Unix shell prompt:

```
% aesop-reset
```

A.2 Reclaiming Space in the Database.

The Aesop system does a poor job of reclaiming space in the database when objects are destroyed. Once in a while (once a month is probably satisfactory), it's appropriate to run the garbage collector to clean out excess space. To do this, run `fcl` at the Unix shell prompt, and at the fcl prompt, type the following:

```
fcl> fam_GarbageCollect
```

The garbage collector will spew a large amount of data as it walks through the database. At the end, it will print a line of statistics showing how many objects it found in its search through the database, how many objects it considered for deletion, and how many objects it actually deleted.

Appendix B. Glossary

Aggregate: An arrangement of components and connectors into a graph structure. An *Aggregate Representation* is a representation of a component or connector as a combination of sub-elements. Also called a *Configuration*.

Asynchronous Connector: A connector used to pass messages in the *Real-time* style, equivalent to a pipe.

Attachment: A joining between a port and a role, indicating that the port plays that role in the interaction defined by the role's connector.

Binding: An identification between a port on a component and a port on a sub-component in its aggregate representation, indicating that the external port corresponds to the internal one. Role-role bindings are similar.

Component*: A* computation or data store of a system.

Configuration: An arrangement of components and connectors into a graph structure. Also called an *Aggregate*.

Connector: An interaction between components.

Design: A representation of a system as a configuration of components and connectors.

Device: In the *Real-time* style, a source of messages.

Input Port: In the *Pipe and Filter* or *Real-time* styles, a port into which messages are sent.:

Filter: A component that transforms streams of data on its inputs into output streams of data with no externally visible state. Used in the *Pipe and Filter* style.

Output Port: In the *Pipe and Filter* style, a port out of which messages are sent.

Pipe: A binary connector that provides sequential delivery of data streams.

Port: A point of interaction of a component with its environment.

Process: A component in the *Real-time* style that transforms streams of input messages into streams of output messages. Processes are loosely akin to *Filters* in the *Pipe and Filter* style.

Representation: A description of the "contents" of a component or connector, either as an Aggregate or as a description in another language.

Resource: A component in the *Real-time* style that represents system resources that are required for a process's execution.

Role: A participant in the interaction defined by a connector.

Sink: A role on a pipe, that attaches to an Input port.

Software Shelf: A repository for architectural elements.

Source: A role on a pipe that attaches to an Output port.

Synchronous Connector: A connector in the *Real-time* style that signifies resource usage.

Appendix C. The Filter Language

The filter language, its structured editor, and its code generator are intended as a demonstration of the strategy of incorporating specialized languages and tools into style-specific environments. The language is not intended to be a useful language in its own right.

The syntax of a filter description is as follows:

> `filter` *Name*
> `inputs:` *decl*
> `outputs:` *decl*
> `static:` *decl*
> `init:` *stmt-list*
> `action:` *stmt-list*

a *decl* is a C decl:

> *decl* `::=` *type* *name* `{,` *name*`};`
>
> *type* `::=` `int` | `char` | `bool`

There are no arrays or structs.

Declarations in the `input` section are input streams. They can only be accessed via the `read` function. Declarations in the `output` section are output streams. They can only be modified by the `write()` procedure.

A *stmt-list* is a list of semi-colon terminated statements. It can begin with local variable declarations as in C. The `init` section is executed when the filter is started; after that, the `action` section is executed repeatedly until one of its sources is closed.

There are three types of statements: assignment, conditional, and write.

> *stmt* `::=` *var* `=` *expr* `;`

Var and *expr* must have the same type.

> *stmt* `::=` `if` *expr* `then` *stmtlist* `else` *stmtlist* `;`

Expr must be a boolean expression.

> *stmt* `::=` `write(` *var* `,` *expr* `);`

Var must have been declared in the `outputs` section, and *var* and *expr* must have the same type. This writes the value of *expr* to the stream represented by *var*.

Expressions are entirely normal. `+` and `-` are the arithmetic operators; they have the same precedence as in C. As in C, arithmetic computations can be performed upon chars. The relational operators `==`, `<=`, `>=`, `!=`, `<`, and `>` all have the same meanings they have in C; they each produce a boolean result. `and`, `or`, and `not` can be used to combine boolean expressions; they have the same meanings as `&&`, `||`, and `!` in C, respectively. Parentheses can be used for grouping expressions as in almost every other programming language.

There is also an expression read(*var*) that takes input from a stream. *Var* must be declared in the `input` section of the filter. The type of the expression is the type of *var*.

The following example illustrates all the features of the language. It maintains an internal buffer of a single character, initialized to 'a'. Given an input stream of characters, it repeatedly reads in a character, and then writes out the lower-case character corresponding to what was previously in the buffer. For example, if the input is ‘Ab?Cd#**eof**’, the output will be ‘aab?cd**eof**’.

```
filter example
 inputs: char producer;
 outputs: char consumer;
 static: char buffer, temp;
 init:
   buffer = 'a';
 action:
   temp = buffer;
   buffer = read(producer);
   if (temp >= 'A' and temp <= 'Z')
     write(consumer, temp - 'A' + 'a');
   else
     write(consumer, temp);
```