Versatile Interaction Specification of Tools and Agents

Rick Neff

UUCS-94-039

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

December 30, 1994

Abstract

Vista is a software infrastructure addressing the vexing problem of software tool interaction—especially how to get *egocentric* tools to work well together. Vista neither assumes nor requires that tools or tool-mediating agents understand a cooperative messaging protocol, only that they share some common means of interprocess communication. Most IPC mechanisms are too ad hoc and low-level for use by non- (or non-expert) programmers. Vista helps by encapsulating such mechanisms in abstract data types obeying high-level protocols. This software framework cleanly integrates a visual language editor, a compiler, libraries, specification analysis tools, and a process control executive into a unified whole.

Copyright © Richard Madsen Neff 1995

All Rights Reserved

ABSTRACT

Vista is a framework and protocol addressing the problem of software tool interaction—getting tools to communicate effectively. Vista neither assumes nor requires that tools or tool-mediating agents understand a cooperative messaging protocol, only that they share some common means of interprocess communication by running under the same operating system or window system, for instance, Unix and X. Existing IPC mechanisms (such as pipes and sockets (in Unix) and (in X) events and selections) are too ad hoc and low-level for enlightened use by non- (or non-expert) programmers. Vista tames their complexity by encapsulating such mechanisms in abstract data types obeying high-level protocols.

Vista is implemented in the Acme Cell Matrix Environment, a graphical, hierarchical, integrated circuit design system. Direct manipulation helps the Vista user "write" an interaction specification in the visual language of cells and wires. Vista's protocol maps cells and their interconnecting wires to computational entities whose spatial relationship largely determines the form and manner of their temporal interaction. In a dataflow sense, wires are value-carrying objects that in a specified sequence enter and leave cells, which in turn are value-manipulating objects. These behavioral cells model process objects of granularity as fine as arithmetic operators, or as coarse as stand-alone application programs. Wires span a similar spectrum from arithmetic operands to inter-application transmission channels.

Four subsystems comprise Vista's framework. The Codifier is a compiler that translates visual specifications into executable code using some C++ class libraries. The Analyzer and the Manifester check the code for syntactic correctness and semantic meaningfulness. The Executer prepares and executes the code thus produced and inspected. This framework cleanly integrates editor, compiler, libraries, specification analysis tools, and process control executive into a unified whole.

To my son, Benjamin

CONTENTS

ABSTRACT iii

LIST OF FIGURES vii

CHAPTERS

1. INTRODUCTION 1
   - 1.1 Problem and Solution 1
   - 1.2 Interface Foretaste 2
   - 1.3 Implementation Preview 4
     - 1.3.1 Object Orientation 4
     - 1.3.2 Hierarchy 4
     - 1.3.3 Direct Manipulation 4
   - 1.4 Formalities 5
   - 1.5 Research Goals 6

2. BACKGROUND 7
   - 2.1 Tools and Agents 7
   - 2.2 The Leaning Tower of Babel 11
   - 2.3 The Worth of Pictures 13
   - 2.4 The Worth of Words 15
   - 2.5 Related Work 15
     - 2.5.1 Interaction as Interface 17
     - 2.5.2 Tcl, the Tool Command Language 17
   - 2.6 Mixed Heritage 18
   - 2.7 More Formalities 23
   - 2.8 Acme 24

3. FRAMEWORK 25
   - 3.1 Acme Object Terminology 25
   - 3.2 Tools, Commands and Features 26
   - 3.3 Protolibs and Prototypes 26
   - 3.4 The Acme Interface to Vista 30
     - 3.4.1 Codifier 32
     - 3.4.2 Analyzer 32
     - 3.4.3 Manifester 33
     - 3.4.4 Executer 34
   - 3.5 Making It So 35
   - 3.6 Summary 35

4. PROTOCOL 37
4.1 To OOP or Not to OOP 37
4.2 Declarations and Definitions 39
4.2.1 Type 44
4.2.2 Order 47
4.3 Practice 48
4.3.1 Making Manifest 58
4.3.2 Parameterization 63
4.3.3 Cooperation 64
4.3.4 Caution 67
4.4 Pushing and Pulling 68
4.5 Finite State Machines 71
4.6 Capitulation and Recapitulation 72

5. APPLICATION 78
5.1 Files, Streams and Pipes 79
5.2 X Synthetic Events 101
5.3 Unix and X United 104
5.3.1 Back Annotation 107
5.3.2 Comments to Code 109
5.4 C++ Comment Stripper State Machine 115
5.5 Sockets and RPC 121
5.5.1 Client to Server 123
5.5.2 Peer to Peer 131
5.6 A Tale of Two Spreadsheets 139
5.7 Mollifying Egocentric Tools 149

6. CONCLUSION 154
6.1 Research Contributions 154
6.2 The Right Choice 158
6.3 The Constancy of Change 163

APPENDICES

A. VISTA CLASS LIBRARIES 165

B. SUPPORTING CODE 224

C. MAKEFILES 240

REFERENCES 262

LIST OF FIGURES

1.1 A Simple Visual Expression 3

1.2 Unix Shell Interpretation 3

1.3 Sample C++ Source Code with Comments 3

2.1 Pipes with T-Junction and File 10

2.2 A Graphical T-Junction 10

2.3 Non-Linear Pipelines 12

2.4 A Computation Engine using a Named Pipe 12

2.5 A Snapshot of the **xgrab** Window 20

2.6 A Synopsis of the **xgrabsc** Program 21

2.7 The **xfilter** Tool in Action 21

2.8 Vista's Eclectic Heritage 22

3.1 The Acme Cell Matrix Environment Graphical User Interface 27

3.2 The Vista Framework 31

4.1 *UnixObject* Class Definition 40

4.2 A Signaler/Signalee Vistafication 50

4.3 A Signaler/Signalee Vistafication with Explicit Vistafier 54

4.4 The **Explain** Main Window 61

4.5 The **Explain** Derivation Window 62

4.6 Vista Specification as State Transition Diagram 73

4.7 Netlist for a Finite State Machine Specification 74

4.8 C++ Code Generated for a *Control* Base Class 75

4.9 C++ Code Generated for *Control* Derived Classes 76

4.10 The *Context* Class for a Word Finder FSM 77

5.1 Ten Commandments for Filters 79

5.2 The *pipe* Class Definition 81

5.3 The `howmanydups` Vistafication 84

5.4 Explicit Dispatch Cells 86

5.5 Sort Deadlock 87
5.6 Wires That Fork *and* Join 88
5.7 The **gnuplot** Schematic 89
5.8 The **gnuplot** Vistafication Preamble 90
5.9 The **gnuplot** Vistafication Body 91
5.10 The **gnuplotvar** Schematic 97
5.11 A **gnuplot**ted Spiral with 5 Turns and 100 Points 98
5.12 A **gnuplot**ted Spiral with 10 Turns and 50 Points 99
5.13 The **hscale** Script in Word and Deed 100
5.14 Pipeline Tapping 105
5.15 Pipeline Splicing 106
5.16 Back-Annotation from **simppl** to Acme 109
5.17 A "Looping" Multifurcated Pipeline 111
5.18 A "Looping" Multifurcated Pipeline with Ordered Joining 113
5.19 The Stripper FSM Schematic 115
5.20 Definitions of Two State and One Action Cell 117
5.21 The **sxe** Client/Server Schematic 124
5.22 Fully-Interconnected Push/Pull Pairs 125
5.23 The **sxe** Wire Type Inheritance Hierarchy 126
5.24 The *WindowEventsStream* Class Definition 127
5.25 Two "sxe" Pusher/Puller Cell Definitions 128
5.26 Automatically Generated Client/Server Headers 131
5.27 Automatically Generated Client Code 132
5.28 Automatically Generated Server Code 133
5.29 The **exi** Peer to Peer Schematic and Vistafication 134
5.30 The **tts** Spreadsheet Schematic 140
5.31 The **tts** Spreadsheet Vistafication 141
5.32 A Bidirectional RPC Integer Exchange 142
5.33 A Unidirectional RPC String Delivery 143
5.34 A Quartet of Cells Interpreting Spreadsheet Commands 144
5.35 The "ttspeer" Agent in Action 148
5.36 Automatic Powerview Project Creation 152

6.1 Vista Versatility Matrix 155
6.2 Modified Stripper Class Definition 160
6.3 Modified Stripper Main Function 160
6.4 Visual Root Dumps, Good and Bad 162
B.1 The "Offer to Save" Dialogue 228
B.2 The Vistafy Command Dialogue 228
B.3 Abort Making "test" in "trial" Dialogue 228

CHAPTER 1

INTRODUCTION

Versatile interaction specification of tools and agents, or Vista, needs at the very outset some definition of terms. The literal meaning of *versatile* is "turning easily, changing or fluctuating readily." Versatile also means many-sided, or multifaceted—having several uses or applications. Versatility is generally considered a desirable attribute. Generally speaking, *interaction* means mutual or reciprocal action or influence. In the current context, it means communication, coordination and cooperation between entities of interest, namely, tools and agents. *Specification* means the act or process of specifying, or making specific. It also means the product of this process, which is a precise and detailed description of something, that something in this case being tool-agent interaction. A *tool* has the usual connotation of an interactive software application program, such as a spreadsheet or a circuit simulator. An *agent* in this context is a program or script that on behalf of a human user drives a tool in the performance of a given task.

This introductory chapter will briefly outline the problem addressed and the solution supplied by Vista. The following chapter will provide some background and motivation. Next follows a discussion of the framework (Chapter 3) and protocol (Chapter 4) developed for Vista. Chapter 5 shows some specific sample applications of the Vista framework and protocol. Chapter 6 summarizes and concludes this report. It is left to the dissertation [67] associated with this report to assess the contributions of this research, and to identify and discuss directions for future research. Specifically, a discussion of the most significant contribution, that is, *patterns* (plus the algorithm for applying these recurring design patterns) will be found only in Chapter 6 (Pattern) and Chapter 7 (Conclusion) of the dissertation.

1.1 Problem and Solution

To see the need for Vista, consider for instance the field of Computer-Aided Design (CAD) of Integrated Circuits (ICs). IC CAD is only one of many application areas where tool interaction and collaboration is a serious problem. Over the years various software tools such as schematic capture editors and logic simulators have been invented to facilitate this increasingly complex task; however, the plethora of tools available to today's circuit designer is more bane than boon. To be useful, tools must be used, but no one tool is appropriate for every facet of the circuit design process, and a compatible and cooperative suite of tools is a pipe dream, or at best a very expensive proposition. Often the product of academic research projects, available (affordable) tools are mostly incompatible, they do not work well together (if they work at all), they rarely "speak the same language." Unfortunately, in most

cases this egocentricity (do it *my* way) carries over and infects their commercial incarnations as well. Successful tool interaction, or *interoperability*, to use a term in vogue (see [61]), requires careful planning and utilization of a variety of interaction idioms.

Vista addresses the very real problem that many interaction paradigms are less than conducive to program coordination, cooperation and communication. The reason for this deficiency is that existing interprocess communication (IPC) mechanisms, such as (in the Unix operating system) files, pipes, streams, sockets, signals, and (in the X window system) events and selections, are rather arcane and too low-level. Programmers must know exactly how to use these mechanisms in order to incorporate them into their programs, which knowledge comes at a considerable cost. To access these mechanisms, the program source code must be modified to include low-level operating system calls or window system library primitives.

Vista lightens the programmer's burden by encapsulating these low-level interaction mechanisms in a high-level framework. This encapsulation results in an inter-tool communication facility that does *not* require a priori that tools adhere to a common messaging protocol. Such tools, whose command language and user interface are vendor-specific or otherwise unique, are deemed *non*-cooperative (or to further anthropomorphize, as above, egocentric)—any interaction with them is always on *their* terms.

1.2 Interface Foretaste

The versatility and flexibility of Vista stem largely from the fact that the interaction specification is in a generalized, visual language. Its generality derives from the underlying organizing principles and computational models implementing this language, which will be introduced later. The visual aspect is motivated mainly by the benefit that entity interrelationships are easier to see.

For example, the Unix shell (command interpreter) language is a textual language, as is C++ [97], although these two languages differ enormously in expressive power. Consider the very simple visual expression of Figure 1.1, which is about as simple an interaction specification of two tools as one can get. The two boxes correspond to the two tools, and the labels inside the boxes identify the tools by name. The line between the two boxes represents their relationship, which though not explicit is quite obvious, at least to the Unix initiate, who will immediately recognize it as a composition useful for answering the question "How many files are there in the current directory?" The neophyte, however, would have no clue as to what it means.

This expression can be rendered quite tersely in shell syntax as shown in Figure 1.2, where there is a one-to-one correspondence between its three tokens ("ls", "|" and "wc") and the three "words" in the visual expression (treating a labeled box as a single word). In Figure 1.3, the same specification in a C++ rendition abandons brevity to be more expressive, and somewhat more indicative of the relationship between the two tools.

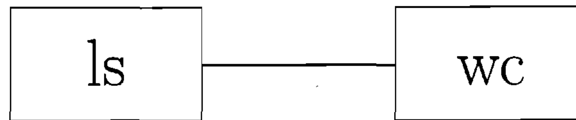


Figure 1.1. A Simple Visual Expression

```
ls | wc
```

Figure 1.2. Unix Shell Interpretation

```
#include "UnixObject.h"    // declares UnixObject class

main() {
  UnixObject pp("pp");     // defines UnixObject called pp
  UnixObject ls("ls", pp); // defines UnixObject ls, writing pp
  UnixObject wc(pp, "wc"); // defines UnixObject wc, reading pp
}
```

Figure 1.3. Sample C++ Source Code with Comments

1.3 Implementation Preview

Vista's visual language and graphical programming environment have been carefully grafted directly into an existing program, Acme (exhibited in Chapter 3), which is well suited for visual programming in that it:

- is object-oriented,
- supports hierarchy, and
- uses direct manipulation.

1.3.1 Object Orientation

Object orientation provides numerous important benefits including data abstraction and encapsulation. Object encapsulation or modularization, which to many software designers is more familiar as the concept of functional or procedural encapsulation, facilitates the separation of interface and implementation, and is especially advantageous in supporting hierarchy.

1.3.2 Hierarchy

Hierarchy, a concept well familiar to hardware designers, is good for reducing cost by sharing storage, and for reducing complexity by hiding detail. Having the interface of an object separate from its implementation facilitates bottom-up or top-down (or middle-out) hierarchical design. In other words, the designer may create an implementation first (bottom-up) or an interface first (top-down). Ideally, the designer should be able to "edit-in-place" both interface and implementation.

1.3.3 Direct Manipulation

Direct manipulation[1] and object orientation go hand in hand. An object has an internal data representation, and an external display representation. The external representation, be it iconic or in structured graphics, can be poked and prodded by the user, moved about the screen, and otherwise manipulated. These "direct" manipulations "indirectly" (transparently to the user) manipulate the internal data structure representing the object, changing its state in some appropriate fashion.

Besides the ubiquitous verb-trio, *point*, *click* and *drag*, another phrase frequently seen in modern window systems is "drag and drop." In many cases, this style of direct manipulation also encapsulates an inter-tool communication mechanism. For

---

[1] The term "direct manipulation" was coined in 1983 by Shneiderman[87], a long-time researcher of human factors and user interface design. Shneiderman was simply giving a name to an already existing phenomenon, since, as he pointed out then, the concept is as old as video games, which have long employed a "joystick" to control and manipulate "objects" such as tanks, guns, spaceships and dot gobbling, disembodied mouths. Games are a natural milieu for allowing the player to directly manipulate the objects of interest, and also for providing immediate visual or aural feedback aimed at maintaining the player's interest and satisfaction. In fact, several human factors researchers ([6, 11, 57]) have argued that, in many cases, "serious" interactive software, in which a mouse or trackball usually displaces a joystick, could benefit greatly from imitating games in this respect.

instance, a user may select an icon representing a file in some graphical directory/file browsing program, "grasp" this icon with the mouse, drag it over the window of another program, and drop it "into" that tool. If this destination program is a text editor, the drag-and-drop gesture may open the file for editing. Similarly, a shell tool may accept the dropped file as an executable program to be loaded and run.

Direct manipulation also goes hand in hand with visual programming; indeed, without it visual programming loses much of its potency and appeal. More detailed justification for adopting a visual approach, along with a discussion of the *disadvantages* of so doing, will be given in the next chapter.

1.4 Formalities

Any serious attempt to raise the level of a language, be it through visual or other means, should take into consideration the formal aspects of languages. In general, formal methods in computer science aim to impress some mathematical rigor on a cavalier discipline, where the crafting of software is still more an art than a science. Formal semantics of a language enables reasoning about, deducing behavior of, and making other logical assertions and proofs about expressions or constructs in that language.

Purportedly, *program verification* produces a formal proof that a program implementation matches its specification. While still a chimera for large-scale systems,[2] the promise of provably correct programs (and "zero-defect software" [31]) is predicated on the premise that submitting to a number of *constraints* gives in return certain *assurances*. For example, enforcing type compatibility between formal and actual function parameters is a constraint that assures the validity of the operations invoked on these data types.

Another noteworthy software engineering paradigm employing some kind of proof procedure formalism is *declarative programming*. Declarative programming differs from the more common *imperative* programming by emphasizing the result, or goal of the program instead of the detailed step-by-step algorithm or method by which that result is obtained. Logic programming à la Prolog is an example of declarative programming [96].

Closely related to declarative programming[3] is the concept of *executable specifications*. Executable specifications imply a way to execute, interpret or simulate the software constructs being specified, usually before committing to a final implementation. The immediate results thus obtained facilitate the rapid prototyping, refining and testing of specifications.

In theory, tool/agent interaction specification is amenable to these kinds of formalizations. In practice, the sheer complexity of these interactions prevents formalizing all but their most superficial aspects. Nevertheless, Vista takes some

[2] In Brooks' eloquent essay on the horrors of the software engineering process [9], he argued that the essential, inherent complexity of software entities will beset and impede progress in this process for years to come, claiming that there is no "silver bullet" that will magically save the computer literati from this monster of their own making.

[3] Some would say identical to, in principle [69].

baby steps in this direction by means of its Analyzer and Manifester modules, which are described in chapters 3 and 4.

1.5 Research Goals

To conclude this introduction to Vista, here follows a brief restatement of the goals of this research:

- Produce a conceptual and an implementational framework, populated with special-purpose libraries and protocols, for tool-agent interaction specification and execution.
- Provide through this framework, or architecture, a means to more productive use of *egocentric* tools.
- Propagate generalized knowledge about this specification/execution architecture and environment.

Implicit in the title of this work is the claim of versatility. How well Vista achieves both vertical (different kinds of tools) and horizontal (different kinds of interaction) versatility hinges on how well it facilitates the productive use of various interaction mechanisms and paradigms for software tools and agents. It is true that any general-purpose programming language is versatile, but to be used productively requires time, effort, and acquisition of the expertise that only comes through experience. On the other hand, trading off versatility for early productivity is characteristic of a more restrictive language or environment, exemplified by spreadsheet programs. Vista seeks the ever elusive happy medium.

CHAPTER 2

BACKGROUND

The first order of background business is a coarse classification of tools and agents, after which follows a general discussion of various existing methods for making them interact. Next come surveys of the literature in visual languages and environments, and of related work in tool integration. Finally, where Vista fits is shown, more on formal methods and models is given, and Acme is introduced.

2.1 Tools and Agents

Recall that tools and agents differ in one important aspect, namely, the fact that tools are designed to be used by human users for general purposes (writing, circuit designing, and so forth), while agents are designed to use tools for particular tasks. The distinction between purpose and task is useful, but need not always be drawn; indeed, it may sometimes be blurred, as in the case of such Unix tools/agents as the shell (**sh** or **csh**), **sed**, **awk**, **perl**, or even **make**. These are programmable tools, *designed* to be commissioned (either on-the-fly or via command scripts) to accomplish sundry and specific editing, text formatting, systems administration or maintenance tasks.

Users employ tools to help them do something that would take much longer and would be significantly harder to do by hand. Agents *represent* users, they stand in for users, automatically doing what users would do manually when using a tool or set of tools. Another conceptual role for an agent is that of a *server* servicing one or more *client* tools, purposefully and intelligently mediating between them. Agents are not "intelligent" in the Artificial Intelligence (AI) sense, that is, no rule or knowledge base, nor inference engine of any kind is involved. All intelligence possessed by an agent is assumed to be imparted to it by the human programmer. For example, the NewWave operating system from Hewlett-Packard (HP) features a user-programmable agent, described by Chew[15] as follows: "The NewWave Agent ... is like a servant who acts for the user. The user writes a Task Language script that is handed to the Agent for execution. These scripts are special objects called Agent Tasks."

It should be acknowledged that artificially intelligent agents are an active area of research. For instance, in [23], Ferguson presents a layered architecture for controlling interacting agents, wherein an agent is defined as "... an autonomous, goal-directed, computational process capable of robust and flexible interaction with its environment." Dealing with sophisticated agents such as these, though interesting, is beyond the scope of this research.

Two main dimensions of tool/agent taxonomy suggest themselves, namely, *static* and *dynamic*. Static taxonomy considers tools and agents as stand-alone objects, describing them as having or not having certain properties or attributes. For instance, the cooperative versus non-cooperative classification mentioned in the preceding chapter is one dimension of distinction using the property of "cooperativeness" as the discriminator. Dynamic taxonomy looks at tools and agents in terms of their mutual *interaction*, naming the various roles they may play. For example, tools and agents may be described as *initiators*, *propagators* or *terminators* of a certain interaction. A given tool may be cast in one, some, or all of these roles, depending on the context in which it is used with other tools.[1]

Continuing with static taxonomy, tools can also be interactive or non-interactive. This distinction is blurred by the fact that all tools have *some* measure of "interactiveness." For instance, while a Unix shell tool accepts interactive commands to run other utilities or tools, the act of supplying command-line options (or parameters) to such tools, or indicating their input sources and/or output destinations, is in a sense interacting with the tools themselves, not just with their execution environment, or the shell. Still, there is a vast difference in both degree and kind of interaction between tools like, for example, **ls** and **emacs**. The former has one purpose in life, to list a directory. The latter suffers from (some say enjoys) the "kitchen sink" syndrome, also known as *creeping featurism*, as it eagerly accommodates ever more functionality through both evolutionary enhancements and its powerful provisions for extensibility (see [94, 95, 27, 30]).

Since most programs of interest consume some input and produce some output, preferably performing some useful computation in between, there is some overlap between the rather broad categories of *producers* and *consumers*. Data is generally consumed or produced in either *synchronous* or *asynchronous* fashion.

These and other attributes are qualitative rather than quantitative. Nonetheless, for many tools quantitative distinctions can be made, in areas such as resource utilization, where, for example, identifying how much memory or how many open file descriptors a tool requires is possible and often necessary to ensure it will run.

Suites of tools *designed to be used together* can cooperate and communicate in a number of ways. A common if cumbersome approach to tool communication is to use an "exchange" data format that each tool can read or write (e.g., the Unix Music Tools at Bellcore [45]). Alternatively, databases can be defined such that tool interaction is done via database access and update operations (e.g., the Octtools system [36]). Where the filesystem serves as the database, the former is actually a special case of the latter.

Another approach celebrates the Unix "small is beautiful" tool philosophy that encourages *composing* small, self-contained, special-purpose tools to achieve more powerful combined functionality (see [43, 75]). This composition is achieved by connecting tools with "pipelines" or lines (links, chains) of pipes. A pipe feeds the output of one tool to the input of another tool, which may in turn pipe its output to yet another tool, and so on for tool chains of arbitrary length. Another view sees a pipeline as a stream of data that is filtered or otherwise transformed by tools

---

[1] For brevity, hereafter read *tools* as *tools* and/or *agents*.

inserted at various places in it. Using pipelines is much more efficient than creating intermediate files between each tool invocation, such that the output of tool n is written to a file that is then read by tool $n+1$, and so on.

Even so, there are circumstances in which intermediate files are both appropriate and desirable. Revisiting the simple example in Section 1.2 to illustrate, suppose the question is "How many of the files in the current directory are mine?" To answer this slightly more complicated question a slightly longer pipeline will suffice, as will a textual rendition of it in shell syntax, as in Figure 1.2, where the vertical bar signifies a pipe connecting the command on its left to the one on its right:

```
ls -l | awk '$3=="neff"' | wc -l
```

In English, this expression says "list the files in the current directory, filter those whose owner (given in field number 3) is *neff*, and count them." A number of digressive observations can be made. Each of the three tools invoked by this shell expression has a rather cryptic name, having been supplied by its creator who no doubt had an aversion to typing. Each tool is given one command line argument, or *option*. The argument to the pattern-matcher **awk** is the pattern to be matched. Using a shell variable (e.g., *user*) would make the pattern more general, not tied to one user. To each of the other two commands is given the same terse option; however, the "-l" does not have the same meaning to **ls** as it does to **wc**. To **ls** it means "long" whereas to **wc** it means "line"—that is, list the files in long (verbose) form, and count the number of lines (not words or characters[2]). Note that omitting the "-l" option to **wc** is okay—it is truly optional. Omitting the option to **ls** is *not* okay—in this context, the option is mandatory for the desired behavior to occur, because only the long file listing supplies the name of the file's owner.

Back to the need for intermediate files, suppose that in addition to wanting to know "How many of the files in the current directory are mine?" the user also wants to know "And by the way, what are they?" A branch in the pipeline to replicate the output of **awk** to a file (named "mine") is an obvious solution, provided by the Unix utility appropriately named **tee**, for T-junction. This is illustrated in Figures 2.1 and 2.2 which show the shell expression and a corresponding visual expression. In this example, the visual expression is in some measure simpler than the textual one, as it exploits the two-dimensional aspect of pictures to obviate the **tee** command.

Now some advantages of visual expressions come into sharper focus. Textual shell expressions are limited to linear chains of pipes, or pipes with T-junctions, and then only files (not more tools) can be attached to the bottom of the T. Arbitrary forking (fan-out) and joining (fan-in) of pipes cannot be expressed, nor can loops where a pipe feeds back into a tool upstream from it. A visual language, on the other hand, can express arbitrarily complex interconnections, as shown in Figure 2.3. In this figure, the labels in the boxes identify the dynamic role or roles each tool plays in this interaction. Thus, 'I' stands for initiator, 'P' for propagator and

[2]Despite its name, **wc** (for word count) by default counts characters and lines as well as words. The "-l" option must be supplied to have it count only lines. In this respect its command line interface is exactly opposite that of **ls**, in that *more* input arguments result in *less* output.

```
ls -l | awk '$3=="neff"' | tee mine | wc -l
```

Figure 2.1. Pipes with T-Junction and File

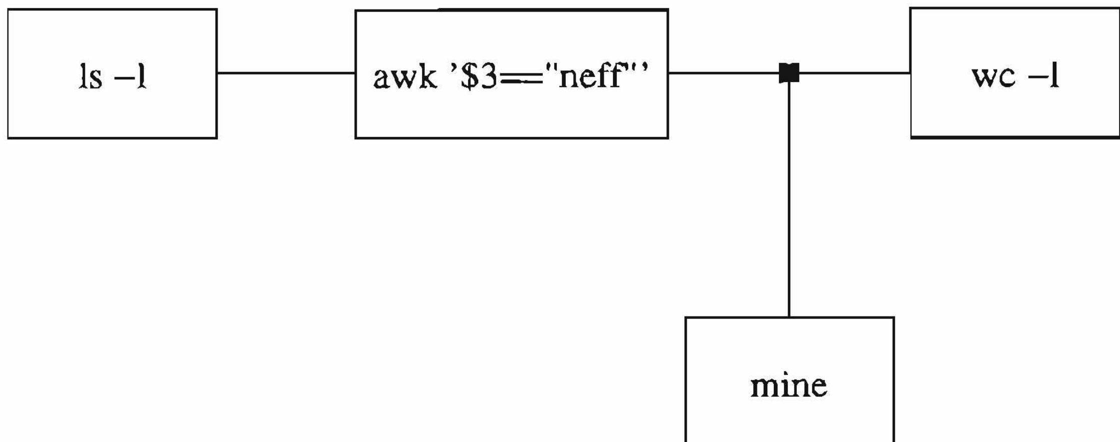


Figure 2.2. A Graphical T-Junction

'T' for terminator. The small triangles denote the direction of information flow, and can be viewed as an arrow head or tail that has been pushed inside the box.

One glaring deficiency of pipes is that they are *one-way* communication channels. One way to effect *two-way* connections between tools is to use a standard pipe in combination with a *named* pipe. Figure 2.4 shows an example of a tool using another tool as a *computation engine.* Tool A creates a standard pipe to tool B, represented by the top unlabeled line. Tool A also creates a connection to tool B represented by the bottom line labeled "BpipeA" to indicate that it is a named pipe. Tool A then pipes commands to tool B telling it to perform some computation, and to write the results to a file. "BpipeA" is the file name A tells B to use. Subsequently, even though it is really a pipe, B can write it and A can read it as though it were a normal file.

A major advantage of named pipes is that they can be used for communication between unrelated processes. Regular pipes require a parent-child or common ancestor relationship between the processes they connect.

For heterogeneous networked computing, Remote Procedure Call (RPC) [68] and its object-oriented counterpart Remote Method Invocation (RMI) appear to be the IPC mechanisms of choice for implementing open Application Programming Interfaces (APIs) for distributed tool communication and cooperation. Client/server dialogues expressed in terms of primitives from such an API library promise to effectively insulate the application programmer from the underlying network communication layers, and thus make remote procedure calls look like local calls. This insulation is not perfect, however, because among other things, failure semantics are significantly more complex in remote calls than in local calls (see [92]).

2.2 The Leaning Tower of Babel

Many highly-interactive tools provide some kind of command language users must master before making productive personal use of the tools, let alone creating and delegating to agents. Unfortunately, these ad hoc languages are in general not well designed, and hence have a clumsy or arbitrary syntax and a steep learning curve. Consequently, users mostly try to get by with learning only a few commands. The potential of such tools is clearly wasted.

Until recently, a Babel of different tools and tool command languages has been the accepted norm, not only in hardware design, but also in software design and other application areas. Recent efforts to ameliorate this situation have been motivated in large part by market pressure—users are no longer simply accepting the norm, but are now demanding that tools from different vendors talk to each other. Though not a vendor, but obviously sensitive to users' needs, the Free Software Foundation includes in its future plans the following desideratum:

> A single command language that could be suitable for use in a shell, in GDB for programming debugging commands, in a program like awk, in a calculator like bc, and so on. The fact that all these programs are similar but different in peculiar details is a great source of confusion. We are stuck with maintaining compatibility with Unix in our shell, awk, and

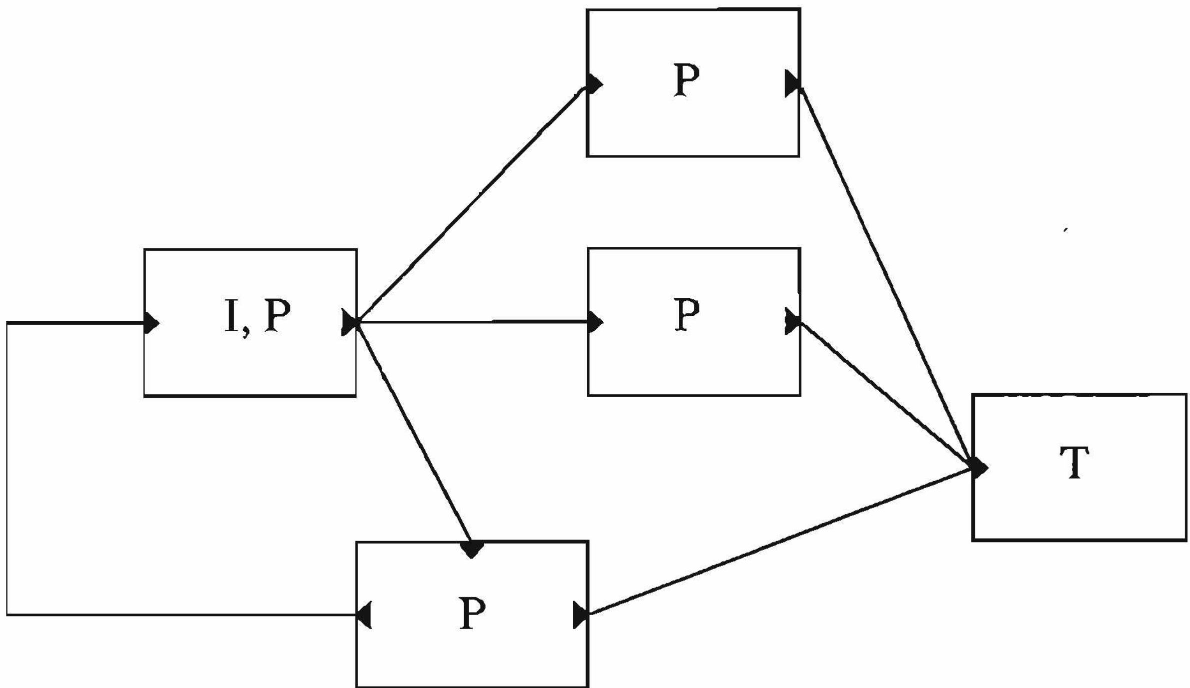


Figure 2.3. Non-Linear Pipelines

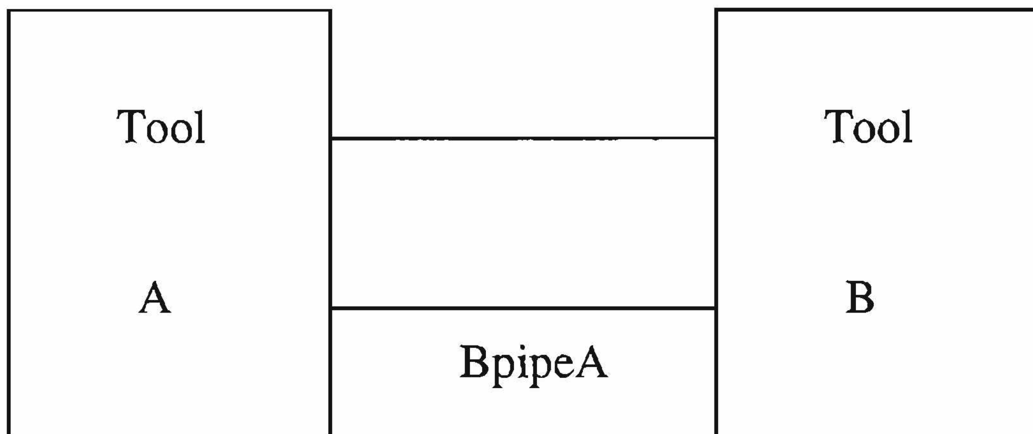


Figure 2.4. A Computation Engine using a Named Pipe

> bc, but nothing prevents us from having alternative programs using our new, uniform language. This would make GNU far better for new users.

In the hardware design arena, languages and inter-tool communication issues are being addressed through standards (such as EDIF and VHDL), frameworks ([37, 56]), and vendor cooperation. Several major CAD tool vendors are currently participating in a cooperative effort called CFI, the CAD Framework Initiative, whose specific charter as stated in [24] is "...to develop industry acceptable guidelines for design automation frameworks which will enable the coexistence and cooperation of a variety of tools."

For software design problems, Computer-Aided Software Engineering (CASE) is currently the popularly prescribed solution[86, 93, 51, 47]. CASE tools and methodologies aim to aid the programmer in every aspect of his or her task, which is to make some abstract computation or computational model concrete using some kind of programming language. Driven by the premise that making the most effective use of the programmer's mind is preferable to making the most effective or efficient use of the computer, higher and higher-level languages are continually being devised. While programs written in high-level languages may be less efficient in terms of time and space requirements than those written in lower-level languages, they are much more easily developed. This is particularly germane to the "occasional" programmer, or end-user-as-programmer, who has neither the time nor the motivation to become an expert programmer, but who has a need to do *some* programming in order to accomplish certain tasks. So-called fourth-generation languages (4GLs[3]) propose to fill this need, as do graphical programming environments.

2.3 The Worth of Pictures

In the areas of visual languages and visual programming environments there is a flurry of research activity, whence more justification for using a visual language for Vista. Chang[14] provides an excellent (albeit a bit dated) survey of research in visual languages, as well as a tutorial in their use. He discusses the concept of generalized icons (object icons or process icons) as a basis for the design of visual languages. In his concluding remarks, Chang designates *dynamic icons* and *icon dynamics* visual paradigms worthy of exploration. Dynamic icons have some time-of-appearance attributes, for example, a blinking traffic light signal that changes its appearance over time. Icon dynamics encompasses the time-sequenced interpretation of an iconic system, where *animation* plays a primary role.

[3] 4GLs are typically non-procedural (declarative) rather than procedural (imperative) languages. That is, they tell *what is to be done*, not *how to do it*. See [4, 58]. For example, a relational database language allows one to say something like "list name and address in employee for salary $>$ 20000" instead of having to supply a step by step procedure that opens an employee database file, considers each file record in turn, filters those whose salary field exceeds 20000, outputs the name and address fields of the filtered records, and finally, closes the file after all records have been processed.

Shu[91] recapitulates nearly a decade of progress in visual programming, distinguishing between visual *environments* and visual *languages*, and also dealing with the concept of *automatic programming*[82, 47]. A representative commercial example of a visual environment supporting automatic programming is National Instrument's LabVIEW[100]. Using this Laboratory Virtual Instrument Engineering Workbench one may visually design software instruments for a broad spectrum of applications, including analytical chemistry, process control, and manufacturing and production. The actual detailed construction of these virtual instruments is performed automatically by LabVIEW based on the visual design.

Representative of research into the nuts and bolts of visual language construction is [18], wherein is described an *expert system* that allows a non-expert user to generate a domain-specific visual language from sample sentences.

From the University of New Mexico comes a visual programming environment called Khoros, described as

> ...an integrated software development environment for information processing and visualization.... Khoros components include a visual programming language, code generators for extending the visual language and adding new application packages to the system, an interactive user interface editor, interactive image display programs, surface visualization, an extensive library of image processing, numerical analysis and signal processing routines, and 2D/3D plotting packages. X applications built using the Khoros User Interface Libraries have built in journal/playback and groupware capabilities.

While quite powerful, Khoros is mainly geared toward image processing and scientific visualization applications, and the process icons available in its visual programming language, Cantata[104], are for the most part large-grained, representing whole programs, or stand-alone application packages.

Closer to (IC CAD) home is Gabriel[46], whose specialized application domain is the design of programmable digital signal processors (DSPs). Gabriel's graphical interface allows the interconnection of icons representing functional blocks, and its block diagram language allows code generation for simulation of high level DSP algorithms. Ptolemy[10], the successor to Gabriel, represents a major effort to generalize the application domain and provide a flexible framework for simulation and rapid prototyping of heterogeneous systems.

Arrays of processors can be designed and simulated with NOVIS[71], whose creators characterize it as a visual *parallel* programming environment. DPOS[21], also a visual language and graphical programming environment geared for developing parallel distributed programs, has the added distinction of supporting *recursive* graphical definition of networks of parallel processes.

Not surprisingly, the visual approach is virtually assured in commercial CASE tools. A major CASE vendor, IDE (Interactive Development Environment) designates its product "Software through Pictures" (see [102]), while another vendor, Serius, invites aspiring software designers to "Get Serius" about graphical and automatic programming. There are many degrees and levels of automatic programming. Several CASE tools implement this concept by generating actual structured code

or code templates from a graphical description, and then allow (or oblige) the tool user (who created the graphical description) to "fill-in-the-blanks" of these generated templates.

2.4 The Worth of Words

As a check on the otherwise unbridled enthusiasm with which visual language proponents promote their ideas, and vendors hawk their wares, it would be well to note the conclusions of Mahling, et al.[55], that the visualization paradigm as implemented in several systems does not necessarily guarantee good human-computer communication, and that graphics alone are inadequate transmitters of knowledge and understanding. They argue for building such visual environments on top of deep representations of the knowledge underlying the relevant problem domain.

Furthermore, reality requires acknowledging other drawbacks of visual languages and visual programming. Even though (or perhaps because) humans have high bandwidth visual information channels, confusion can easily result from visual clutter, or pictorial information overload. One cannot use a simple text editor to create or modify visual programs. They are in general hard to formalize. For these and other reasons, it appears that text will always be needed in some form or other—what is desirable (albeit elusive) is a salubrious mixture of text and graphics, such as is attempted by the LIVE[44] language. The authors of this **L**anguage for **I**ntelligent and **V**isual **E**nvironment aim to show that "visual programming languages and textual programming languages are not opponents but can complement each other."

To achieve this complementary relationship, LIVE allows visual data objects to be "picked" (selected) either by *position* on the display screen using the mouse, or else by *name*, using the keyboard for input. The latter is often preferred by experienced users as it is more efficient when many objects crowd the screen and more precise when positional picks are ambiguous. Compatibility and coupling between visual representations and textual representations give LIVE preciseness and generality, qualities often found lacking in visual languages [89].

Other mergers of the visual and the textual include such techniques as "prettyprinting" program source code, which employs visual formats, indentation, spacing, font type and size variations, and so forth to clarify the logical structure and meaning of the text, resulting in easier reading comprehensibility. An example of an application program that not only prettyprints code, but also acts as a text previewer, database browser or menu utility, is Lector [78] from the University of Waterloo. Lector takes descriptively marked-up text as input, and provides flexible text display and interaction by *structuring* the text to distinguish between its content and its visual display or layout characteristics.

2.5 Related Work

This section discusses past and current work more directly related to tool interaction and integration, necessarily presenting only a cross-section of a megalith of models and mechanisms.

IShell[7] is a visual programming environment for Unix. Based on the dataflow metaphor, its iconic command language, IScript, allows users to connect icons

representing Unix tools/utilities and visually depict the flow of information between these tools.

The FIELD programming environment[80, 81] integrates tools by the mechanism of selective broadcasting, whereby a message server reroutes incoming messages to specific clients that have registered interest in them. Tools cooperate by exchanging messages in one of three modes:

1. point-to-point (one-to-one),
2. multi-cast (one-to-some), or
3. broadcast (one-to-all).

The SunSoft ToolTalk Service[17], a commercial derivative of the FIELD message server, adds an object-oriented message passing system to the more procedural messaging modes enumerated above in what is touted as a step toward the future "Distributed Objects Everywhere" environment.[4] Billed as an "open" inter-application communication facility, ToolTalk enjoys the distinction of having been chosen as the underlying messaging service of the CAD Framework Initiative[26].

Whether procedural or object-oriented, ToolTalk messages come in two flavors, *notices* and *requests*. Likewise, message recipients are identified as either *handlers* or *observers*. Handlers handle requests; observers notice notices. A request blocks the sending process until a handler responds, while a notice is non-blocking, allowing the sending process to continue processing without waiting for a reply.

SPARCWorks from SunPro uses ToolTalk to integrate its suite of software development tools, which includes a tool manager, a "make" tool, a debugger, a source browser, a file merger and a performance analyzer.

SoftBench from HP integrates its suite of comparable tools via TIP, the Tool Integration Platform, also providing an "Encapsulator" for plugging non-cooperative, third-party tools into the system, allowing them to communicate with the existing tools without source code modifications[41]. This is very similar to the CFI's Tool Encapsulation File facility, which provides a veneer of common syntactic forms for specifying how tools are invoked, the types of command-line arguments it takes (boolean switches, strings, filenames, etc.), and other hooks for on-line documentation and iconic representation in a graphical user interface (GUI) framework.

HP is also a strong proponent of the Common Objects Request Broker Architecture (CORBA) specification defined by the Object Management Group (OMG) consortium, which has over 300 member companies. CORBA [20, 73] specifies the mechanisms by which objects may transparently make requests and receive responses over a network. For instance, one of these mechanisms is DII, the Dynamic Invocation Interface, by which clients may dynamically compose and invoke object messages and requests.

---

[4]Project DOE, as it is called, envisions the eventual happy marriage of object-oriented technology with heterogeneous networked computing environments.

2.5.1 Interaction as Interface

A class of tools known as "user interface builders" (UIBs) promises "high productivity programming" solutions to the onerous problem of human-computer interaction, of which tool-agent interaction is but a subset. These tools provide help for the tedious and error-prone tasks of designing and programming the user interface for a given application. Most work by allowing the user to directly manipulate graphical elements representing user interface elements such as menus, buttons, scrollbars and dialogue boxes. Placing these elements (sometimes called "widgets") creates a mock-up of the application interface. Code for implementing this interface is automatically generated by the tool.

Garnet[65] is a gargantuan UIB for constructing highly interactive user interfaces. Garnet provides many interesting modes of interface behavior specification, including the highly visual "programming by demonstration" mode, whereby a language of button presses and mouse gestures can be translated into executable code. Also known as *programming by example* (PBE) this mechanism has strong automatic programming overtones, and is being touted as the next step beyond direct manipulation [64]. Other less ambitious but still interesting UIBs include the following, which are noteworthy if only because all three are freely available:

Ibuild [99] is a UIB for the InterViews [53] toolkit, which is a library of user interface components layered on top of the X window system [84] and implemented in the C++ programming language. An example of a user interface built using Ibuild is that of the **Explain** program. (See Subsection 4.3.1.)

XF [19] uses **wish**, the windowing shell based on Tk (pronounced "tee-kay") which is an X toolkit based on Tcl (pronounced "tickle") (see Subsection 2.5.2 below) that allows one to write entire applications as Tcl/Tk scripts, without the need to resort to C code. XF embeds itself in the script it generates, enabling modification of the application's interface *while the application is running.*

WAFE [70] (Widgets (Athena) Front End), also based on Tcl but using the Athena widget library instead of Tk, decouples the user interface from the application by providing mechanisms for allowing the front-end user interface and the application program to run as separate processes, communicating via standard Unix input/output streams.

2.5.2 Tcl, the Tool Command Language

The Tcl language deserves special attention. Having adherents who promote it with missionary-like zeal, Tcl[72] is both a shell scripting language (with variables, lists, loops, procedures, etc.) *and* an embeddable interpretive command language with a Lisp-like flavor.[5] More specifically, calls to the Tcl *library* can be embedded in applications written in C (or C++) to give them a built-in string-based command interpreter. Although it has a somewhat different syntax, like Lisp Tcl treats code

[5]However, Tcl is much more lightweight than Lisp—for one thing, C-style strings are Tcl's *only* data type.

and data the same. That is, by manipulating string data representing the body of a procedure, new procedures can be constructed and defined on-the-fly in the Tcl interpreter, making the command language extensible by the application. The guiding principle behind Tcl is that a single interpretive language should control all aspects of all interactive applications, including:

- Function of,
- Interface of,
- Composition of (pieces of), and
- Communication between applications.

In a best-case scenario, Tcl would be the command language used by all applications. With this ideal level of cooperation, the Babel of idiosyncratic command languages would cease. Huge, monolithic tools of necessity providing a superabundance of functionality would become a thing of the past. In their place would be lean, efficient, specialized reusable tools that all communicate with each other, using the **send** primitive provided by the Tk library.[6] Any tool could send a Tcl command to any other tool, invoking internal functional and/or external interface actions. Moreover, the user interface to these tools could be *built* interactively (using a Tk-based tool like XF) and also *changed* dynamically as the application is running. Such is the power of a single interpretive language.[7]

One notable example of how Tcl has been used successfully to control interactive programs programmatically is **expect** [48]. Designed to overcome certain limitations of standard Unix shells,[8] **expect** is a high-level shell language with the full power of Tcl at its disposal. Scripts written using **expect** essentially contain a multiple-path dialogue between two or more interactive programs. Control flow is managed via send/expect sequences, which specify commands to be *sent* to an interactive process, what to *expect* in reply, and what to do when the reply matches one of the expected patterns. Not only is simultaneous control of multiple programs possible, but control can also be passed back and forth from **expect** to the user, in effect allowing the user to be treated as an interactive source or sink of I/O.

2.6 Mixed Heritage

How does Vista relate to all of the above? An example or two may begin to motivate the answer. It was stated in Subsection 2.5.1 above that tool-agent interaction is a subset of human-computer interaction. Given the propensity of humans to tinker with tools, it might be more accurate to say that these two

[6] This presupposes that all of these tools are running on top of the X window system.

[7] Naturally, such power comes with a performance price. Furthermore, application composition is necessarily coarse-grained.

[8] Manifested by such programs as **passwd** that *demand* to be invoked interactively and hence cannot be controlled by a shell script via I/O redirection or pipes.

endeavors have a non-empty intersection. In other words, there is overlap but not containment. Building a graphical user interface as a friendly front-end may or may not be viewed as a natural first step in the evolution of a *human-driven* tool into an *agent-driven* tool.

For example, the **xgrabsc** tool allows the capture, or "grabbing" of a screen or window in X, and then saving the image to a file or sending it to a printer. Using this kind of tool one can take "snapshots" of window contents—textual, graphical or both—for later reproduction or processing. As **xgrabsc** has a ridiculously large number of command-line options, an interactive front-end called **xgrab** was implemented to afford easy selection of the various options, and push-button invocation of the **xgrabsc** program. Figure 2.5 contains a snapshot of the **xgrab** window (made by **xgrab**, of course) showing its option-setting "buttons" and option-editing text-input fields. The corresponding command-line options of **xgrabsc** as given in the Synopsis section of its Unix "man" page are reproduced in Figure 2.6 for comparison.

Even an interface like **xgrab** loses its effectiveness if the need arises to capture dozens of windows for further processing. An agent programmed to control **xgrabsc** directly (or even indirectly via **xgrab**) is definitely called for. Likewise for another interactive interaction specification tool called **xfilter**, whose user interface is shown in Figure 2.7. A simple interactive application for filtering data through Unix commands such as **tr**, **spell**, etc., **xfilter** uses the X *selection* mechanism[9] and maps selectable input and output locations to the standard input and output of a Unix filter command. The input/output panel contains option menus for setting the data source and output target. The default input is "Primary selection" and "Text window" is the default output. Thus, in the figure the selected text (taken from a window where the text of this paragraph was being viewed) was passed through the filter "tr [A-Z] [a-z]" and then deposited in the scrollable text window. The source and target can also be files, making **xfilter** useful as a data transfer facility between X windows and Unix files.

While **xgrab** and **xfilter** were not implemented using a UIB, they easily could have been. But UIB built or not, they are simply inadequate for *unattended* tool interactions. Unlike Vista, UIBs are mainly geared to specifying interactions primarily with human users, neglecting for the most part communication between software entities.

In short, while there is a kinship between Vista and UIBs, as well as between Vista and the other tools and environments mentioned above, this relationship does not imply redundancy. In true eclectic fashion, Vista borrows ideas from all of these[10] as it strives for free and fluid versatility in the kinds of interaction specifications it supports. This will become more clear in the following chapters as the Vista framework and protocol are unveiled. For now, the void Vista tries to fill is depicted in Figure 2.8.

---

[9]The Inter-Client Communications Conventions Manual (ICCCM) dictates how X11 applications must use this and other interaction mechanisms [85].

[10]Indeed, nothing prevents Vista from directly and advantageously *using* **expect** or **awk** or **perl** or **make** or even **emacs**, etc.

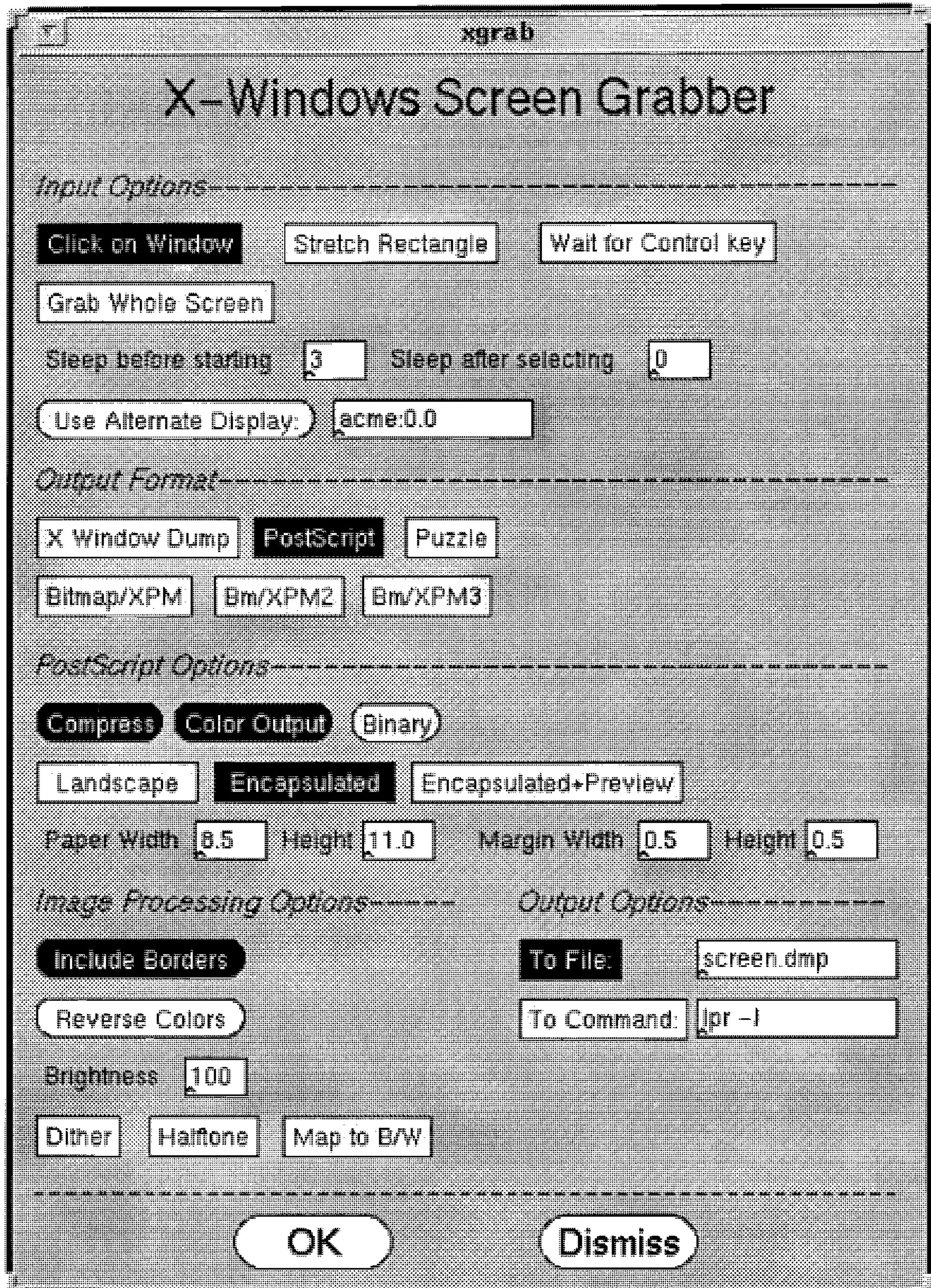


Figure 2.5. A Snapshot of the **xgrab** Window

```
xgrabsc [-d display] [-id windowId] [-o outputFile]
        [-s seconds] [-post seconds] [-b percent]
        [-and andBits] [-or orBits] [-page widthxheight-hmarg-vmarg]
        [-bell -grab -verbose -bdrs -nobdrs
         -key -stretch -root -click
         -reverse -bw -mdither -dither -halftone -nodither
         -ps -cps -simple -xwd -bm -bm2 -bm3 -puzzle
         -bin -comp -eps -l -limit -preview -prev -colproc]
```

Figure 2.6. A Synopsis of the **xgrabsc** Program

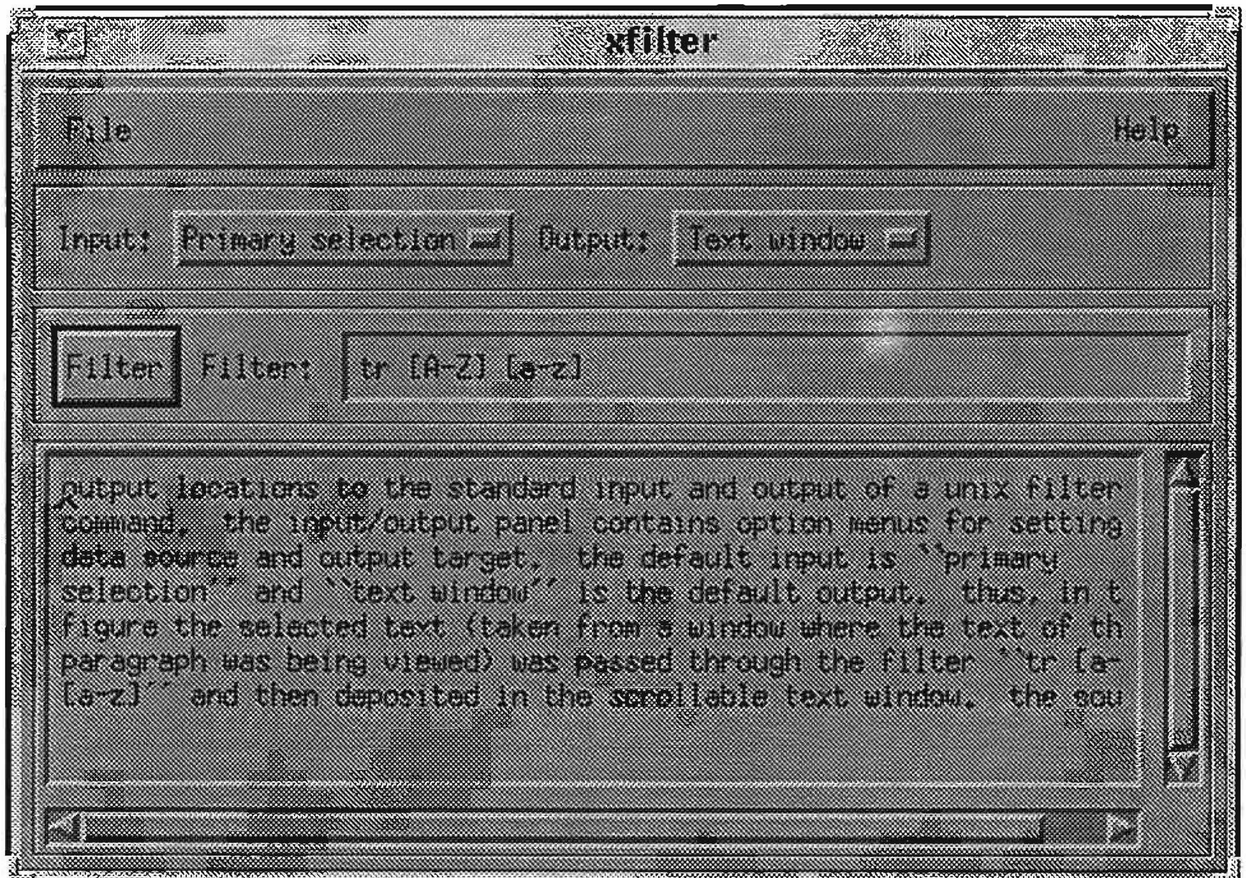


Figure 2.7. The **xfilter** Tool in Action

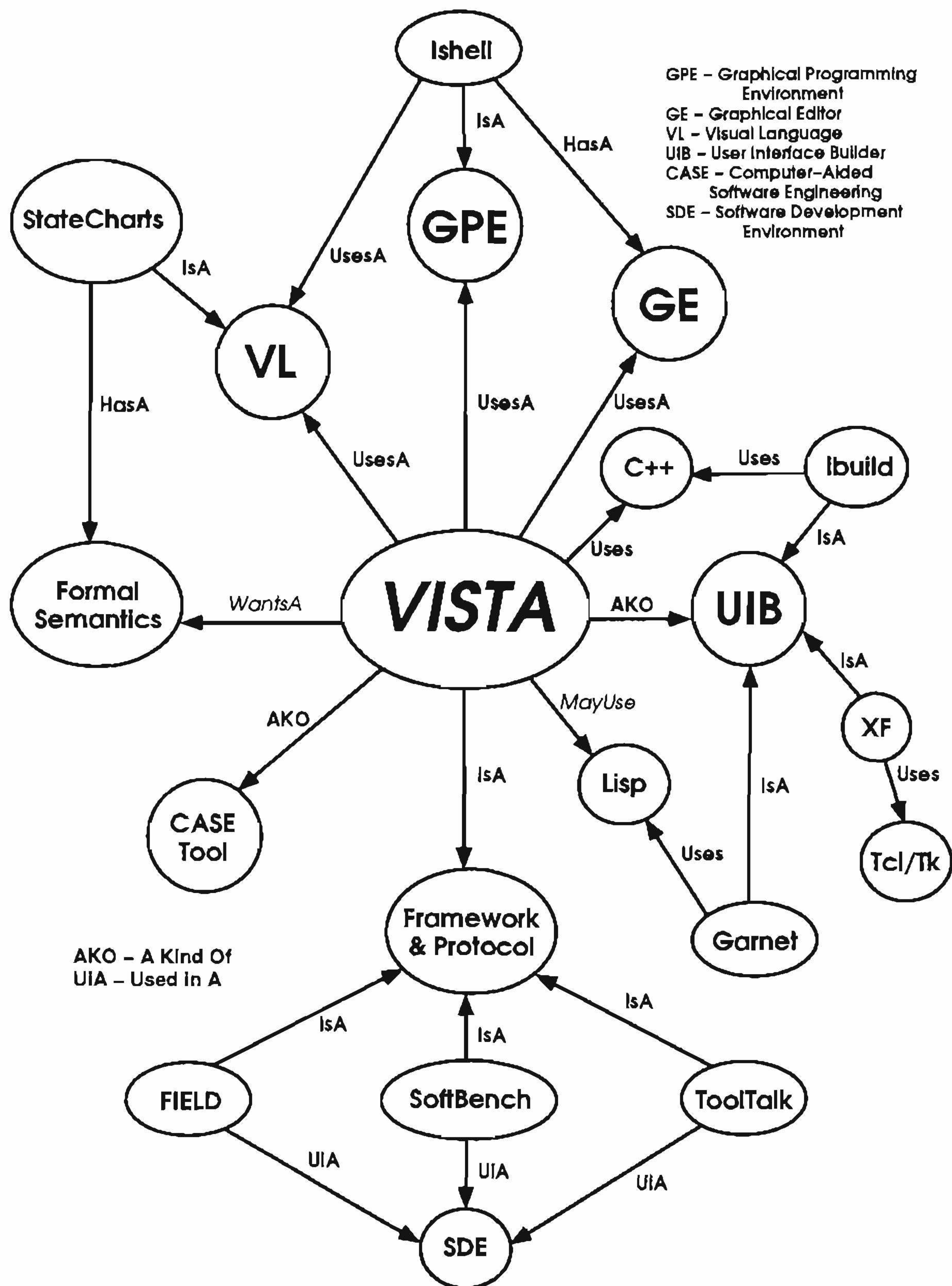


Figure 2.8. Vista's Eclectic Heritage

2.7 More Formalities

A number of computer science luminaries have tackled the Herculean task of bringing to this discipline the attribute *formal*, long exemplified by mathematics, to fortify the *experimental* or the *observational*, exemplified by physics. No less a luminary than Robin Milner, inventor of ML (MetaLanguage), has most recently grabbed the limelight. In his Turing Award Lecture, Milner[62] presented his contributions of the Calculus of Communicating Systems (CCS) along with the π-calculus, which make *interaction* the cornerstone of an algebraic calculus.

Other attention-commanding work on formalizing models and specifications is set forth in Harel's *Biting the Silver Bullet: Toward a Brighter Future for System Development* ([34]). Prompted by a desire to illuminate the brighter side of the gloomy picture painted by Brooks' position paper ([9]), in this article Harel responded with his own thoughtful vision of the future. In it, the concepts of *executable models* and *visual formalisms* [33] rise to the fore. The former were introduced briefly as *executable specifications* in Section 1.4 above. The latter attempt to make system modeling a predominantly visual and graphical process by finding appropriate visual representations for various conceptual constructs, and exploiting properties such as containment, connectedness, and adjacency, all of which have formal, mathematical counterparts from set theory and topology. It is noteworthy that in the final chapter of [90] (which discusses future prospects for visual programming as it moves toward formalism) Harel's concept of visual formalisms (as realized in his *statecharts* (and commercialized in the **statemate** environment) [32, 35]) was held up as a maturing development in the field.

Extending traditional state transition diagrams to handle the notions of hierarchy, concurrency and communication, statecharts provide a way to describe reactive behavior that is simultaneously clear (i.e., comprehensible to humans) and formal (i.e., amenable to computer manipulation and analysis).

Users of PegaSys[63] can graphically and formally describe the hierarchical structure of programs, revealing their coarse- or fine-grained data and control entity dependencies by means of *formal dependency diagrams* (FDDs). A number of dependencies at one level of the hierarchy can be composed into a single dependency considered atomic at a higher level. It is also possible to do the inverse operation of top-down decomposition.

As with statecharts and (possibly) FDDs, highly reactive systems (e.g., physical hardware) are the ones that benefit the most from the Petri Nets[74] model, which has a long and successful history of use for systems modeling and analysis (e.g., [79, 3]). This powerful and useful model has a well-developed formal mathematical theory, supporting rigorous analysis of systems composed of separate, interacting components exhibiting concurrency and therefore needing synchronization.

While it is not within the Vista purview to do general-purpose visual programming or behavioral modeling of complex, reactive systems, the kinds of visual formalisms that are appropriate for these are also relevant to Vista—for instance, "zooming" capabilities for moving between levels of abstraction. Moreover, the event-driven nature of reactive systems makes them analogous to agent-driven tools in the sense of needing to react to external and internal stimuli—granted, the former at fully or nearly continuous rates, the latter much more sporadically.

2.8 Acme

The Acme Cell Matrix Environment is a graphical, direct manipulation interface to the PPL Cell Matrix integrated circuit design methodology[12, 28], with several extensions, the major ones being *domain integration*—the support of hybrid (mixed physical and structural) design[40]—and the ability to juxtapose several different design contexts, such as analog and digital. Acme is a successor to INSTED[66], and as such, seeks to remedy the numerous flaws of its predecessor.[11] Acme is written in C++ and uses InterViews to implement its GUI. A more detailed description of Acme is deferred to the following chapter.

It was chiefly the desire to do mixed-mode, mixed-level simulation of circuits designed using Acme, with the idea of controlling the simulation from within the Acme environment, that originally motivated the provision of mechanisms for interfacing Acme with several disparate simulation environments. In general, interaction with a circuit simulator requires:

1. a description of the circuit in a form it understands (a netlist),
2. a description of the stimulus (input vectors or waveforms) to the circuit,
3. a sequence of simulator commands, and
4. a disposition of the results, that is, what to do with the output of the simulator.

Given the code for netlist generation currently residing in Acme, an interface of sorts to various simulators already exists. The exchange of information is achieved by reading and writing files. Acme can generate the netlist file, but the onus is on the designer to write command script files and read simulator output files. Without the benefit of back-annotation, where, for example, wire values are shown directly on the design schematic, interpreting simulation results is quite burdensome, as the designer must mentally or manually make the association between reported simulator outputs and the output nodes in his or her design.

Because the granularity of communication links can be much finer, sophisticated design-tool inter-communication à la ToolTalk is clearly superior to the old-fashioned file-based approach; however, its use is precluded when trying to get non-cooperative tools to communicate. In this case, the need is clear for encapsulating or mediating agents that talk a certain way to the non-cooperative tools, and possibly another more efficient way to each other.

An interesting opportunity afforded by the dual use of Acme as design tool and Vista framework is to both specify and execute the interaction in the same environment. Thus, Acme becomes just a tool interacting with other tools, in this case, simulators. The agent mediating between them can not only use a special cooperative protocol to transfer simulator results and annotations back to Acme, but it can also communicate to Acme the current status of the *interaction* itself. This allows for the Vista specification to be animated in some fashion corresponding to its execution, by highlighting or otherwise identifying activity hot spots.

[11] With 20-20 hindsight, the wisdom of Brooks[8] is readily acknowledged.

CHAPTER 3

FRAMEWORK

This chapter describes the Acme Cell Matrix Environment, discussing the features and benefits that make Acme suitable as a visual programming environment in general, and as the implementational framework for Vista in particular. Prefacing this discussion is a terse glossary of some Acme terminology, and following it is a presentation of the pragmatics of the Acme/Vista visual language, and an exposition of the Acme/Vista interface.

3.1 Acme Object Terminology

cellmatrix — a container for cells, wires and vias.

cell — an instance of a prototype containing local state.

prototype — a template for a class of cells (a cell master).

protolib — a collection (or library) of related prototypes.

wrapper — a graphical view of a prototype (for customizing cells).

sticker — like a wrapper, but more lightweight (for customizing wrappers).

port — a point on the edge of a cell at which a connection may be made.

wire — a connector of cells by way of ports.

The properties and terminology associated with cells, ports, wires, prototypes and protolibs will be expounded as needed to develop the concepts embodied by Vista. For the moment, to elaborate on the visual aspects of wrappers and stickers, conceptually these are "wrapped" around (or "stuck" on) cells, giving them a customized look at various levels of abstraction. Symbolic or name wrappers are more abstract than gate or transistor wrappers, the former hide detail, the latter flaunt it. While none have been done to date, a *documentation* wrapper could be used to display (and store) a textual description of a cell's structural or behavioral semantics. As they allow for different graphical representations of the same object, wrappers not only supply the basic iconic vocabulary for Vista's visual language, but they also give it variety and the ability to express nuances. Wrappers and/or stickers (along with modifiers, which are local state objects associated with a cell), are also useful for implementing dynamic icons à la Chang (see Section 2.3).

3.2 Tools, Commands and Features

In the style of several widely-used drawing editors (MacDraw and its many clones), Acme has two orthogonal user interaction mechanisms, *tools*[1] and *commands*. A tool defines a context, or mode for using the mouse. First the mode is engaged, and then the mouse can be used to invoke an operation. By contrast, a command is a stand-alone operation, taking effect immediately when invoked (although it may call for further dialogue). Tools embody the verb-noun interaction style, that is, a tool (verb) acts on an object (noun). The orthogonal noun-verb style of commands implies first selecting the object(s), and then specifying the operation to be applied to the selected object(s). Figure 3.1 shows Acme's GUI with the tool panel on the left side and the command menu bar at the top.

Besides the direct manipulation paradigm embodied in its objects and tools, Acme holds a strong commitment to other established principles of good user interface design[88]. For example, a keystroke is associated with each tool and most commands, providing the shortcut demanded by experienced users. This is actually more important for commands than for tools, as the latter enlist the mouse for pointing, clicking, dragging, and other operations that have no convenient keyboard accelerator.

Another user compassionate[2] feature Acme implements is a generalized history mechanism, which provides the means for any operation that makes a change to the cellmatrix to be undone. For instance, for operations that insert cells, the Undo command deletes these cells and restores what was there previously, if anything. Likewise, undoing a delete operation will put back what was deleted. Redo is the convenient counterpart to Undo.[3] These commands work by means of "change objects" that maintain just enough information to restore the cellmatrix to its previous state. A specialized change object class is defined for each kind of undoable operation. Change object instances are stored on a history list of arbitrary (user-settable) length limited only by available virtual memory. Chang's icon dynamics (Section 2.3) are realizable by means of history list traversal.

3.3 Protolibs and Prototypes

Retrofitting Acme with a visual programming environment was a relatively easy task, due in part to the feasibility of making extensive use of the data structures and direct manipulation mechanisms already in place. Far harder was the creative work of building various special-purpose Vista protolibs, and populating them with "abstract" prototypes, which is to say, prototypes having behavioral but no physical implementations. Such prototypes possess the usual interface consisting of ports

---

[1]These tools should not be confused with the self-contained tools that are the primary focus of this research.

[2]That is, more than friendly. See [2].

[3]More precisely, Undo undoes the last change made to the cellmatrix, doing nothing if all stored changes have been undone. Redo redoes the last undone change made to the cellmatrix, that is, it undoes an Undo. Redo does nothing if it follows a "Do", i.e., any command that *does* something.

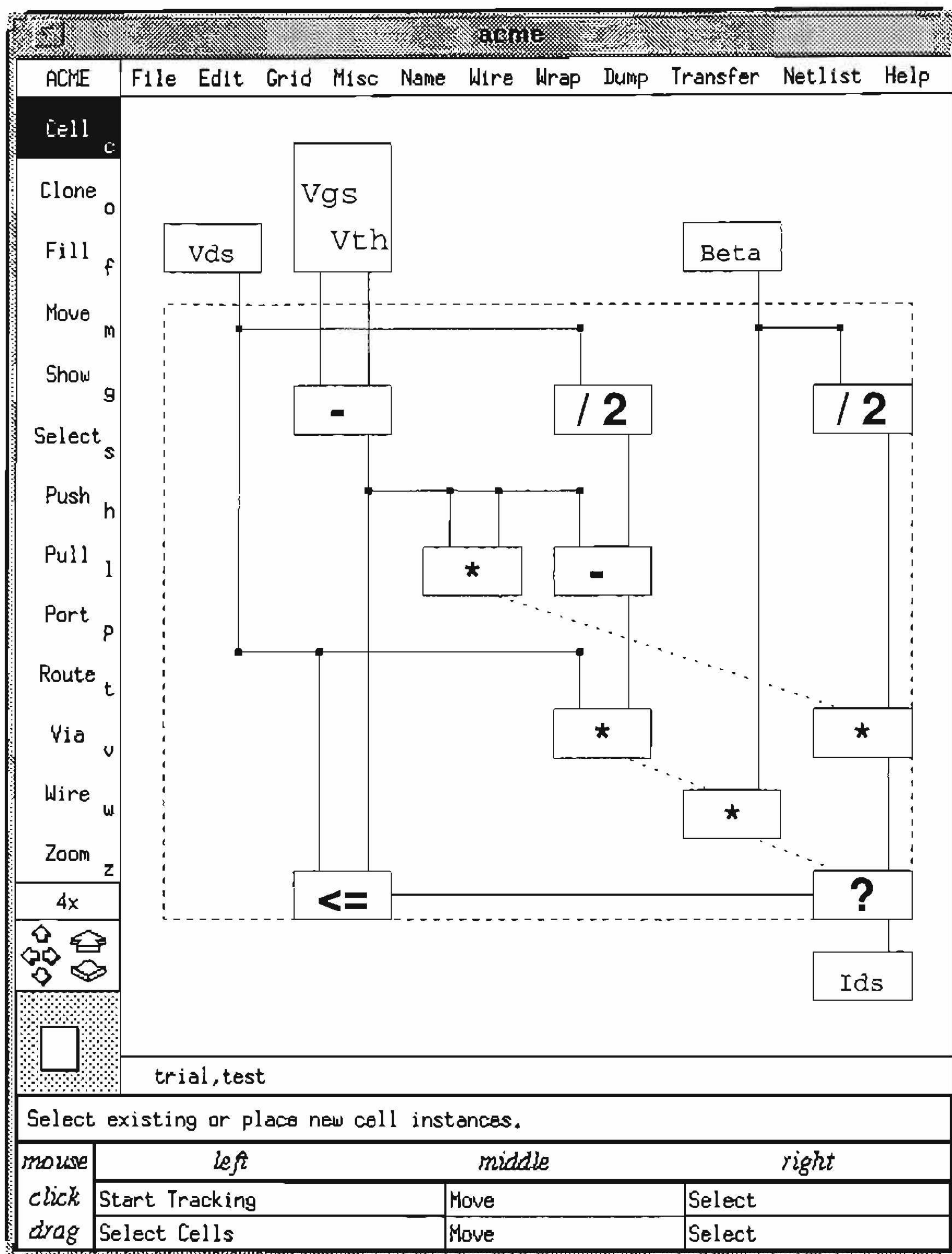


Figure 3.1. The Acme Cell Matrix Environment Graphical User Interface

and a shape, which, because the prototypes are abstract, can be the smallest unit-cell shape regardless of the complexity of the behavior they encapsulate. They must have *some* shape to be stored in a cellmatrix, but this need not be the shape or size presented visually. Indeed, the wrappers defined for these prototypes can be of arbitrary size, and can even overlap, although they typically do not.

The abstract behavioral implementations of Vista prototypes may be defined textually or graphically, or both, and these implementations may be edited, extended, interchanged and parameterized, as may the interfaces of these prototypes. The idea is to provide a large degree of flexibility to the Vista library designer, within the bounds of the Vista conceptual framework, which bounds are necessary to ensure the coherence and integrity of Vista specifications.

The following list itemizes in approximate chronological order the principal Vista prototype libraries, each of which encapsulates a different interaction mechanism, or group of related mechanisms:

- **sig** — signals (Unix)
- **fsp** — files, streams and pipes (Unix)
- **xse** — X synthetic events (X Windows)
- **unx** — Unix and X united (see paragraph below)
- **cts** — Client to Server RPC
- **ptp** — Peer to Peer RPC

These libraries will be described more fully and several examples of their use given in the following two chapters. Note that these Vista libraries can be unified by using the *protolib referencing* feature Acme provides. For example, the **unx** library references **fsp** and **xse**, making their combined services available in a single protolib context.

Given these libraries of prototypes having parameterizable interfaces and implementations, the pragmatics of Vista's visual language are as follows. Using Acme tools and commands, Acme objects are created and manipulated. Cells are interconnected with wires via ports. Cells, ports and wires are all named objects, identifiable and accessible by name. Collections of cells and wires may be "wrapped up" to create new hierarchical prototypes, whose semantics are defined more or less by the composition of their constituents. There is virtually no limit to the number of levels of hierarchy allowed, although one to three usually suffice.

Once wrapped, new instances (sometimes called *hiercells*) of non-leaf-level prototypes (called *cellmatrix* prototypes) can be created and inserted into the cellmatrix design. If desired, the details of their internal structure can be laid bare at any time by the operation known as *clear wrapping*. Also available is the inverse operation of *opaque wrapping* to once again hide these details. These are read-only operations as far as the cellmatrix prototypes are concerned, since only the local *wrap* state (clear or opaque) stored in the hiercell is changed.

Once a network of cells and wires has been created, the user invokes the Acme "Vistafy" command, which provides the actual user interface to the four Vista

subsystems, which are briefly introduced now, and described in more depth later on in this chapter:

1. Codifier
2. Analyzer
3. Manifester
4. Executer

Briefly, the Codifier takes the interconnection network of cells and wires and from it generates stylized C++ source code whose form resembles a netlist.[4] The Analyzer and the Manifester verify and explain syntactic correctness and semantic meaningfulness (or lack thereof) while the Executer has the task of actually creating the executable specification, and spawning it from within the Acme environment.

Correct syntax demands compatibility of cell interconnections, which is checked using Acme's port typing mechanism. A *PortType* object is associated with each port in a prototype's interface. Port types attach behavioral semantics to ports, and thence to the wires that connect them, by identifying the data types of the "signals" that will flow through wires into cells by way of their ports. Analogous to data types in programming languages, these port types can be statically checked for compatibility at "compile time," (or even before that at "edit time," that is, as the ports are being wired together). Implemented as a specialized *NamedObject*, the PortType has a name attribute maintained as a character string that matches the name of a predefined class type in a Vista protolib.

Compatibility is not restricted to (same name) equality, but rather means membership in a family of compatible types, which are grouped hierarchically in parent/child relationships. Read in from a text file stored in the protolib directory, the in-memory (directed acyclic) inheritance graph of port types is created and maintained in the Vista protolib. The protolib has methods for determining if two or more port types are compatible by querying this inheritance graph. See Subsection 4.2.1 for specifics.

To guarantee correct semantics, Vista protolibs have certain rules that must be followed. For instance, ports have *modes* in addition to types. The three modes currently defined are *input*, *output* and *biput*, which describe the direction of data flow into or out of cells. Thus, one rule to check is that input ports must have *at most* one wire attached to them. This restriction may be removed by defining a "fan-in" semantics for input ports. Another rule is that all ports must have *at least* one wire attached, which also may be relaxed for ports that represent optional parameters. It is the concerted work of the Analyzer and the Manifester to ascertain adherence to these rules, and to show the user what is wrong (or right) with a given Vista specification.

[4]Examples of this code, along with the behavioral models Vista associates with cells and wires will be presented in the following two chapters.

3.4 The Acme Interface to Vista

Figure 3.2 shows the Vista Framework as currently implemented in symbiotic relationship with Acme. The salient features of this implementational framework are greatly abbreviated in this entity-relationship diagram, which therefore calls for some explanation.

The square box containing the two labeled rectangles represents the heart of the Acme/Vista symbiosis. The small solid squares on the edge where arrows touch the outer square suggest the contact points (or communication ports) for this software framework, representing both the GUI, with its tools and commands, and the software interface to the external environment. The lower rectangle is subdivided into four smaller rectangles, each with at least one dashed edge shared with its neighbor. Not surprisingly, the four single-character labels stand for the Codifier, Analyzer, Manifester and Executer, which, as the dashed edges suggest, have no hard boundaries between them. In fact, these four Vista modules are tightly connected and somewhat interwoven, as will be seen. The four black square ports between them and Acme are perhaps misleading, because there are *not* four distinct well-defined interfaces between each module and Acme, but rather, as stated above, one interface—the Vistafy command. The fact that each of the four modules necessitated some new functionality being added to Acme may justify the one-to-one correspondence. At any rate, the idea that users use (or interact with) Acme directly and Vista indirectly should be clear. But, as the other arrows leading from the rounded box labeled *Users* suggest, there are other interaction paths available to users. Users may write their own classes, or Vista class libraries, as long as they adhere to the Vista protocol (see next chapter). Of course, if desired, they may also read the code for the various classes comprising the Vista core protolibs, as it is purely textual C++ code. Furthermore, those classes must be read by, and, if creating new classes, initially *written by*[5] Acme/Vista in the process of generating the executable specifications, which are the special-purpose *agents* that when executed provide the runtime liaison between Acme and other tools.

The relationships labeled *generates* and *executes* between agents and the Codifier and Executer, respectively, are the critical ones. The Analyzer and Manifester play important, but subordinate roles, and, to avoid cluttering the figure their relationships (both internal and external) are not shown. The bidirectional *talk to* relationship between agents and tools is colloquial for communication, or interaction. Note especially the direct access to tools and agents available to users. If they wish, users may invoke the Vista-generated agents directly, as after all, they *are* executable programs. Likewise, if desired, knowledgeable users can interact with tools directly without going through agents. The middle arrow, between users and the *talk to* arrow, is perhaps the most interesting possibility. Eavesdropping on a "conversation" between agents and tools is sometimes useful and illuminating.

[5]This is a bootstrap operation, strictly for user convenience. Vista writes a skeleton class definition corresponding to a newly-wrapped user-defined prototype, which the user must subsequently flesh out.

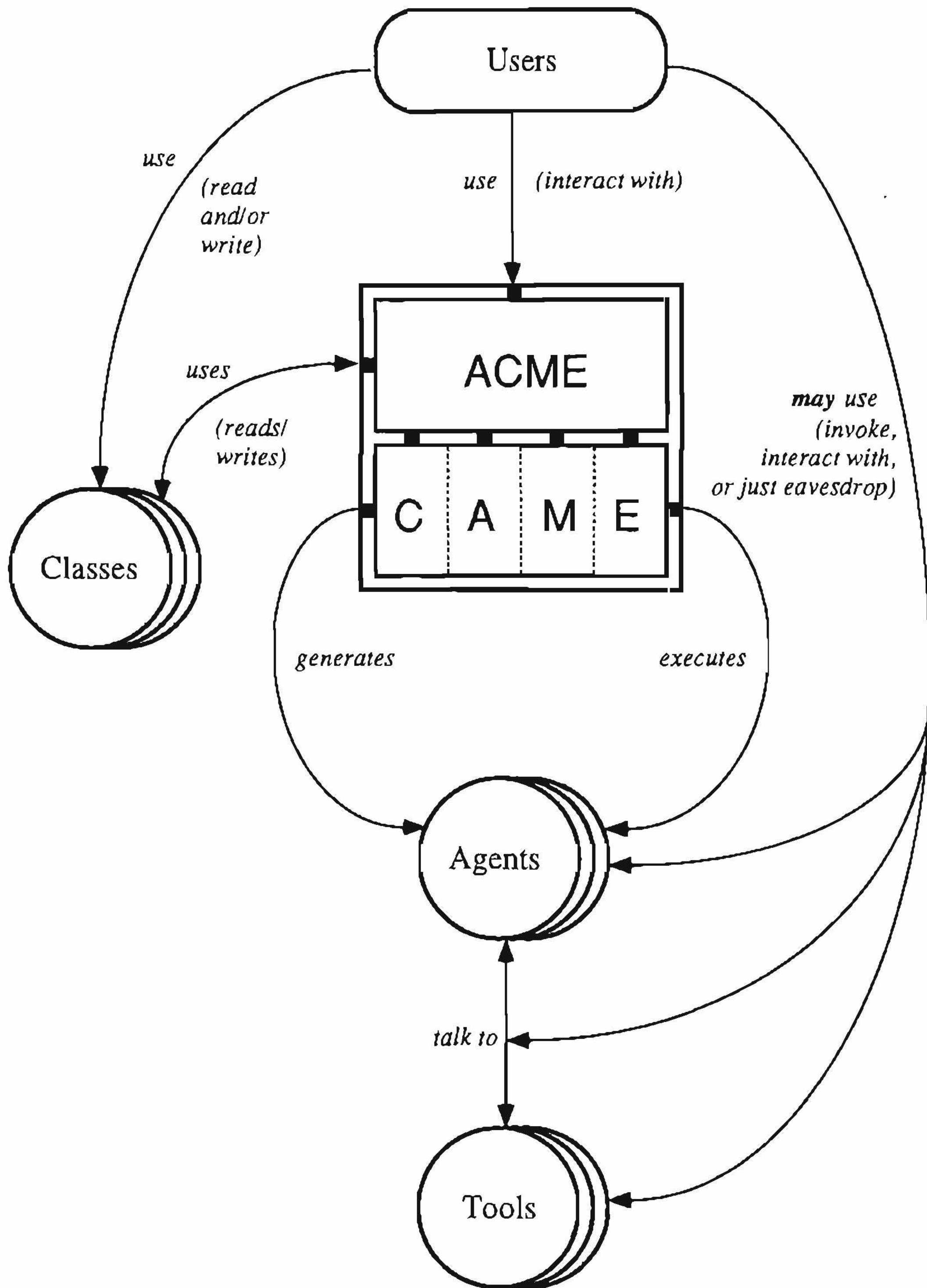


Figure 3.2. The Vista Framework

The specific ways this may be accomplished will be examined later, after laying the appropriate groundwork.

Included in Appendix B, in addition to the C++ code itself, is an English description of the Vistafy command, enumerating the steps necessary to produce, analyze and run an executable Vista specification. Each of the four subsystems controlled by this command has its own subsection below, even though, as mentioned above, these four modules are very interrelated.

The name of the cellmatrix design containing the user-created Vista schematic is used as the base name for several ancillary and intermediate files that are produced by the Vista subsystems, as well as the final result, which is the executable agent. This executable C++ program has the identical name as its source cellmatrix, that is, no file extension is appended to it. In the following discussion, assume `test` is the name of this source cellmatrix schematic.

3.4.1 Codifier

The code implementing the Codifier is distributed throughout several Acme modules and source files. The reason for this lack of locality is that the Codifier is essentially producing a netlist of cells and wires, and the already written netlist generation code is likewise distributed, in turn due to the many intricate class interdependencies found in Acme.

To accommodate a variety of simulator netlist formats, Acme uses a table-driven approach to netlist generation. A table is associated with each separate netlist facet, including formats of file headers and trailers, sub-circuit definitions, sub-circuit calls, and so forth. For each supported netlist format, a string containing formatting directives is stored in each of these tables, which are indexed by an integer representing the simulator type. Adding support for Vista thus required little more than adding a new simulator type[6] and inserting the Vista-specific formatting strings in the appropriate slots in the tables.

The netlister traces the connectivity of cells and wires by traversing (at each level of the hierarchy) the arrays, trees and tables containing these objects, which are maintained by the cellmatrix object, writing out the behavioral models, wire declarations, sub-circuit definitions and sub-circuit calls to the netlist file, which for Vista is a C++ `.c` file. When hierarchy is involved, Vista by default generates *structural* models, meaning that sub-circuits are defined by what nested structure they contain, not by a strictly *behavioral* model.

For each wire, the cellmatrix looks for all cell ports that are physically coincident with the wire, and infers the type of the wire from the types of all ports so found. This type inference is germane to the Analyzer, and may be considered part of its work, whence the blurring of the line between it and the Codifier.

3.4.2 Analyzer

Should the type determination just described reveal that incompatible ports are connected, instead of reporting the error immediately, the fact is noted by giving the

[6]At a few strategic points in the netlisting algorithm, it was also necessary to add some conditional statements, applicable only to the Vista simulator type.

wire a spurious type, namely, a concatenation of the names of all the incompatible types, separated by a special marker. When the Analyzer scans the `.c` file produced by the Codifier it looks for this marker, and writes any bogus types it finds there to a `.z` file for later use by the Manifester.

Another task well suited to a textual pre-scan of the `.c` file is the discovery of ports without wires attached to them. In certain special cases this may be allowed, but in general, it signifies a connectivity error. As with incompatible-type detection, when the Codifier is navigating the connectivity graph of cells and wires, and is ready to write out the sub-circuit calls, one per cell, it must include in each call an ordered list of the wires attached to each port on the cell. On finding no such wire, the deferred-notification approach it once again takes is to write out the name of the *port* prefixed by another special marker to identify it as an unwired port. These go into the `.c` file, where the Analyzer looks for them, but it puts the corresponding *cell* name into the `.z` file, rather than the names of the unwired ports on the cell. The Manifester takes it from there.

3.4.3 Manifester

The Manifester has the mission of making manifest to the user the errors discovered by the Analyzer during the attempted compilation engaged by the Codifier. This approach is preferable to letting the compiler catch such errors as described above, which it surely would were they not intercepted by the Manifester. The compilation is only begun if the `.z` file exists and is empty, which means that the Analyzer has examined and validated the `.c` file. If non-empty, there are only two kinds of entries in the `.z` file. Details of these entries, plus a description of how the Manifester notifies the user of these errors, are found in the technical report.

Having shown the user the syntax errors of his ways, the work of the Manifester is not yet finished. As intimated above, semantic errors can also be detected and reported, given a sufficiently sound database of facts and rules, together with an inference engine that can deduce whether or not the facts play by the rules.

Generally speaking, facts summarize certain cardinality and connectivity features of a given specification, while rules express various cardinality and connectivity constraints. Detailed examples of some facts and rules will be given in the following two chapters, but an important point to emphasize here is that, unlike for finding wrongly-typed wires and unwired ports, perusing the text of the `.c` file is a clumsy and inefficient means of extracting the facts, and is wholly inadequate to the task of establishing the rules. The assumption Vista makes is that the Vista protolib designer creates the rules database explicitly, and implicitly arranges for the companion facts database to be created *during the execution of the specification.* What this implies is that a significant part of the Manifester is embedded in the behavior of the cells and wires in a Vista protolib.

The selection of a suitable inference engine was not easy, due to the superabundance of freely (as well as commercially) available software packages. The package chosen was developed at the University of Wisconsin-Madison. As it "COmbines Relations And Logic" in one deductive database/logic programming system, **coral**[77] possesses many nice features that make it an attractive and useful ally to the Manifester, including:

- A declarative language based on Horn-clause rules using Prolog-like syntax.
- Extensions like SQL's group-by and aggregation operators.
- Support for many evaluation techniques, including bottom-up fixpoint evaluation and top-down backtracking.
- A module mechanism, providing for separate compilation and the use of different evaluation methods within a single program.
- Support for data types including numeric and string constants, functor-terms, lists, sets, multisets, arrays and non-ground terms.
- An "explanation" facility that allows users to examine "derivation trees" for facts using a graphical menu-driven interface.

Once it has created the facts database, the executing **test** program itself then calls upon *system* to `make OUT=test manifest`. The actual Vista "makefile" is included in the TR, but briefly, what making manifest does is to produce a file suitable for submission to **coral**, whereupon it runs **coral** in batch mode[7] with this file as input. The file contains commands instructing **coral** to consult both facts and rules, and to use its explanation feature to show the user the result of evaluating these databases, which answers the "Is everything okay?" query.

3.4.4 Executer

If everything *is* okay, the Executer uses the Unix *fork/exec* system-call pair to create a child process in which to execute a successfully analyzed and compiled `.c` file. As specified by this paradigm, the forked child process begins life as an exact clone of its parent process (Acme), but can then customize its own copy of the parent environment before it mutates (via *exec*) into a process running the compiled (e.g., **test**) program. These customizations include putting a new variable (VISTALOG) into the environment, whose value the newly executing process will read from *its* environment, upon which it opens for writing a file by this name (which would be `test.a` in this example), and at well-defined junctures, logs certain useful information to this file. Before the *exec* call, other data is conveniently passed from parent to child by way of the argument list given to *exec*, such as the "X Window Id" of both the main Acme window, and its *CellMatrixView* subwindow, which serve to identify the proper recipient of synthetic X events, needed by the **xse** protolib (see Section 5.2).

The Executer is considered to have begun its work when the Codifier forks a subprocess in which to make the **test** executable. That is, the compilation of a Vista `.c` file is part of the preparation process of making a specification executable. It should be pointed out that a viable alternative to compiled C++ may be an *interpreted* target language, which would obviate the compilation step. This alternative is discussed in Chapter 7 of the dissertation.

---

[7]It is possible to use **coral** interactively as well.

3.5 Making It So

The important role played by the Unix **make** tool in the Executer, Manifester and Analyzer modules makes it worthy of a few brief observations here, and a lot more exposure hereafter (also, see [22, 25] for more on **make**).

The ostensible purpose of the **make** tool is to determine automatically which pieces of a large program need to be recompiled, and invoke the commands to recompile them. Not limited to compilation, however, **make** can be used to describe and perform *any* task where some files (called targets) must be regenerated from other files whenever these other files (called dependencies) are changed. For example, the first thing the Vistafy command does is what **make** would do if told that `test.c` depended on `test.cm` and how to generate the former from the latter. Invocation of the Codifier, however, is best handled directly by Acme rather than (very) indirectly by way of a **make**-issued shell command.

Highly parameterizable (particularly the implementation furnished by the FSF, which has file inclusion, conditional execution, and a host of other features), the various aspects of **make** are as:

- a dependency graph builder,
- a target updater, and
- an executable specification (the *makefile*).

The makefile is really a database that contains a description of the dependency relationships among many files, plus the rules (shell commands) for updating the target files from the files they depend on. Querying this makefile database, **make** determines which files need updating by consulting the facts "database" consisting of file last-modification times. For each "out-of-date" file, it executes the specification (issues the update commands) stored in the database. The functionality afforded by **make** makes it an excellent tool-mediating agent, for which assertion more evidence will be furnished in the next two chapters.

3.6 Summary

Vista is implemented in the Acme Cell Matrix Environment, a graphical, hierarchical, integrated circuit design system. Direct manipulation helps the Vista user "write" an interaction specification in the visual language of cells and wires. Acme essentially serves as a graphical editor for creating specifications.

The encapsulation of an Acme cell prototype, with its separation of interface and implementation, facilitates bottom-up (implementation first), top-down (interface first), or middle-out design of hierarchical, behavioral cells. Cell prototypes reside in a protolib, associated with which is a library of C++ classes, one per cell. Wire (port) types are also implemented as C++ classes.

Four subsystems comprise Vista's framework, or infrastructure. The Codifier is a compiler that translates visual specifications into executable C++ code using the class libraries. The Analyzer and the Manifester check the code for syntactic correctness and semantic meaningfulness. The Executer prepares and executes the code thus produced and inspected. This framework cleanly integrates editor,

compiler, libraries, specification analysis tools, and process control executive into a unified whole.

CHAPTER 4

PROTOCOL

The word *protocol* is laden with several meanings, especially in computer science where it has a somewhat different connotation than it does in, for instance, diplomacy. This chapter explores its meaning in the domain of object-oriented programming (OOP) in general, and in Vista in particular.

4.1 To OOP or Not to OOP

In object-oriented programming, a class is defined to have certain operations or methods for dealing with objects of that class. The set of operations thus defined is conceptually the *communication protocol* these objects understand and obey. The semantics of these protocols serve to limit inter-object dependencies, since in general an object can only access the internal state of another object through the "public" methods of its protocol, or interface. This encapsulation decouples objects, isolating them from one another, which promotes modularity and reusability. Further benefits flow from class inheritance and polymorphism. Objects derived from a common base class (thus inheriting a common protocol) can be used without knowing to which specific derived class they belong. The C++ virtual function mechanism implements a form of *dynamic binding* (also called late or runtime binding) whereby functions redefined by the derived class (subclass) are automatically invoked on these objects, as determined at runtime by their specific types. Virtual functions redefined in subclasses "shadow" or override their corresponding base class functions, and thus enable subclass-specific behavior at the same time the common protocol makes possible the *uniform* treatment of common base class objects. In essence, the details of the class derivation hierarchy are hidden by the encapsulation enabled by this runtime type resolution. (See [54], chapter 9.)

For example, recall the Acme change objects mentioned in the previous chapter. Each undoable Acme operation has a specialized class defined for it by subclassing the base *ChangeNode* class; for instance, *DeleteChange*, *MoveChange* and so forth, all inherit the *ChangeNode* structure and protocol, redefining its virtual methods, in particular *Undo* and *Redo*, while adding the necessary state peculiar to that type of change. The history list contains *ChangeNode* pointers only, and uniformly invokes *Undo* or *Redo* on *ChangeNode* objects, which translates dynamically to the appropriate *Undo* or *Redo* method associated with the actual *MoveChange*, *DeleteChange*, or whatever object these pointers really point to.

The basic principles of object-oriented programming briefly outlined above have been catching on in the world of programming with ever-increasing momentum. Marketing hype notwithstanding, they constitute a major paradigm shift for soft-

ware development. Yet this shift is resisted by a number of programmers who for various reasons are hesitant to adopt the "everything is an object" mentality, and are reluctant to buy into OOP as a panacea.

Their leeriness is well-founded. For one thing, the jury is still out on whether OOP is the tidal wave of the future or merely a passing fad. For another thing, everything is *not* an object, or put more correctly, not everything is an object. *Actions* are things too; otherwise, language would have no use for verbs. While it is possible and in many cases desirable to conceptualize and implement actions (or other abstractions) as objects, in general the actions or operations attached to objects in OOP are not themselves objects (instances of a class), and thus become, in a sense, second-class citizens.

Furthermore, the OOP mandate that state and behavior be fused in objects flouts the fact that not all behavior is ascribable to one or another distinct, easily defined class of objects. That is, some behavior is decidedly *inter*object, especially when disparate objects are interacting, and as to which object to bind this behavior to, the decision is arbitrary. Should the bread be buttered, or the butter breaded? More often than not, the behavior is attached to neither of such a pair of objects; instead, a third party, a coordinating *agent* acts on both. This agent, which is akin to but finer-grained than the agents defined in Chapter 2, manipulates the state of each object through the objects' defined interfaces, but the manipulation, the behavior itself is not part of those interfaces. Bread and butter are passive objects, acted on by an *active* object, a person (presumably with a knife, another passive object) in the action called spreading. The real world has yet to see self-spreading butter, or self-buttering bread, although the world of software has no quarrel with such anomalous, autonomous objects.

Another deficiency in extant OOP languages is their inability to articulate constraints on method invocation, such as the conditions under which methods can or must be invoked, valid sequences of invocation, and so forth. These constraints should be considered part of the protocol, but while they may be documented by the programmer via comments placed directly in the code, or elsewhere, the language itself offers no means of expressing them. Eiffel[60] actually does a better job than C++ in supporting the attachment of *pre*conditions and *post*conditions to method code, the violation of which triggers an elegant *exception handling* mechanism. True, exceptions are now officially part of the C++ language, but many implementations are poorly (or non-) compliant with the defined standard. At any rate, exceptions are only useful for error *detection* and subsequent recovery—rescuing executions gone awry because the protocol, the interobject contract, has been broken somehow. They do nothing for error *prevention*—ensuring the integrity of the protocol—something the proverbial *sufficiently clever compiler* could conceivably do at compile time, given a sufficiently rich knowledge of valid object relationships, interaction constraints and operating assumptions. One possible way Vista may be used to model such knowledge is outlined below in Section 4.5.

As was noted earlier, actions *may* be cast as objects, and one way to do that is to define a class whose methods implement a given action or set of actions. These methods operate on objects other than the one to which they are bound by the class mechanism. How best to partition a large system into objects and actions is by no means an exact science, and indeed may be dictated primarily by aesthetics. In any

case, along with function binding in object protocols goes the notion of *delegation*, which is to say, whether and to what extent an object does the work or hands it off to another object, possibly but not necessarily of the same type. This allocation of responsibility is really a part of the protocol, or *meta*protocol, now applied not only *intra*object but also *inter*object.

A caveat before proceeding: the following refinements to the term *delegation* are not found *per se* in the OOP literature. Indeed, there is a possibility of confusion between its usage in this chapter and how it is used in other object-oriented languages, e.g., the classless (*prototype*-based) *Self* language[98, 13]. That said, with inheritance, the explicit invocation of a base class method from a derived class method is termed delegation to a *superior*, that is, higher (more general or basic) in the derivation hierarchy. A full delegation, where all the work is done by the base class, is redundant, as base class methods not redefined in the derived class are invoked implicitly. Partial delegation makes more sense in that part of the work is performed by the base class, while the rest—presumably applicable only to the more specific subclass—is done by the subclass method. In addition to inheriting structure or protocol from a base class, objects can *contain* other objects, in which case invocation of a contained class method from the container class method may be designated as delegation to an *inferior*. Object methods can also refer to other objects passed as arguments, and invoke methods on them as delegation to a *peer*.

4.2 Declarations and Definitions

A closer examination of Figure 1.3 may make some of these abstractions more concrete. At first glance, the code in the main procedure looks like it does nothing but declare three variables, *pp*, *ls* and *wc*—a declarative specification, no more. At second glance, noting (per the comments) that these variables are objects of type *UnixObject*, the declarations look more like definitions—definitions that take arguments, no less. In fact, this code relies on the automatic object initialization provided by the C++ language to do its work, to make this specification *executable*. A special method known as a *constructor* is associated with each class. As its name implies, the constructor is called to *construct* a valid object of its class in the storage allocated for it, whether on the heap or on the stack. The three *UnixObject* instances are all built on the stack; the first by a constructor that takes a single argument (a character string), and the second and third by constructors taking two arguments. Like any other C++ functions (or operators), constructors can be *overloaded*. That is, the same name can be used for different functions, the only requirement is that the functions' argument lists be different. Each overloaded function is then distinguishable from its other namesakes by having a unique "type signature" given to it by the C++ compiler. Thus, the third constructor differs from the second by passing its arguments in reverse order.

Figure 4.1 presents in skeleton form the class declaration and protocol of *UnixObject*, which inherits from a parent class, *VistaObject*, indicated by the first line:

```
class UnixObject : public VistaObject
```

```
class UnixObject : public VistaObject {
public:
  UnixObject(const char* name) : VistaObject(name) {
    CreatePipe();
  }
  UnixObject(const char* name, UnixObject& uo) : VistaObject(name) {
    CreateProcess(name);
    AttachOutput(uo);
    RunProcess();
  }
  UnixObject(UnixObject& uo, const char* name) : VistaObject(name) {
    CreateProcess(name);
    AttachInput(uo);
    RunProcess();
  }
  void AttachOutput(UnixObject& uo) {
    // either delegate to uo peer, or ...
  }
  void AttachInput(UnixObject& uo) {
    // ... just access pipe state in uo
  }
  void CreatePipe(void) {
    // using the Unix ``pipe'' system call
  }
  void CreateProcess(const char* name) {
    // using the Unix ``fork'' system call
  }
  void RunProcess(void) {
    // using the Unix ``exec'' system call
  }
};
```

Figure 4.1. *UnixObject* Class Definition

The *public* keyword indicates that all publicly visible (accessible) data and function members of the base *VistaObject* class are also visible to clients (users) of the derived *UnixObject* class. In this hypothetical protocol, the task of the first *UnixObject* constructor is to create, via a low-level system call, a Unix *pipe* object that can join two Unix *process* objects, whose creation is handled by the second and third constructors. In addition, the second "attaches" the writing end of the pipe to the newly-created *ls* process, while the third attaches the reading end of the pipe to the *wc* process. Both pipe and process objects are assumed to be encapsulated in the *UnixObject* class. The details of this encapsulation are suppressed for simplicity, but it could be that *UnixPipe* and *UnixProcess* class objects are created and delegated to, either as superiors (using *multiple* inheritance) or (more likely) inferiors (data members contained within a *UnixObject* instance). Though not explicitly shown, it also could be useful for each *UnixObject* constructor to delegate to its *VistaObject* superior certain bookkeeping responsibilities, such as registering the name passed to it as a key in a global look-up table mapping names to objects.

While it may appear in this sample code that a performance penalty due to excessive procedure call overhead is inevitable, with the function *inline* feature of C++ this is not the case. The inclusion of class member function bodies within the defining scope of the class is typically sufficient to advise a C++ compiler to expand calls to that function inline, with appropriate parameter substitution, in effect stripping off the usual procedure-call encapsulation.

As hinted to by the above use of the adjective "hypothetical," this is not the actual protocol Vista uses. There is in fact no *UnixObject* class, nor is there a *VistaObject* class for that matter. In the course of its evolution, an organizing principle has arisen in Vista. This principle posits that there should be not one, but *two* fundamental types of objects, with fundamental differences in the ways they are created and used. The two primitive entities in Vista were cursorily introduced in the previous chapter. To reiterate, they are cells and wires, represented graphically by boxes and lines, respectively. Relationships between these entities are represented by their interconnection, as in any graph or network, but also by the connection *type*, as imposed on wires by cell ports.

The distinction between cells and wires is maintained principally by convention. Exceptions are not disallowed (e.g., see Section 4.5), but conventionally, cells have no mutable state, they only act on wires, which *do* have mutable state. In this sense, cells are like procedures in most traditional action-oriented languages, whereas wires are like the data structures operated on thereby. Thus may Vista be accused of taking a giant step backwards by separating rather than combining state and behavior. However anomalous this may seem, in Vista both cells and wires are bona fide objects, instances of *subclasses* of the Vista base classes *Cell* and *Wire*—an interesting case of using OOP to implement pre-OOP abstractions. As it turns out, doing so is simply more convenient from a programming standpoint.

Procedures usually take parameters, and in Vista, ports represent *formal* parameters, whereas wires map to *actual* parameters. The port name as used in the body of the procedure associated with a cell stands for the mutable wire object this body of code will act on when invoked. A wire attached to a cell is automatically typed according to the type of the port at which it is attached. The kinds of operations allowed in Vista cell procedures are unlimited, as C++ *copy* constructors

and operator overloading permit parameters and operands used in expressions to be of *any* user-defined class type, not just the primitive built-in types, such as *int* or *float*. A copy constructor is needed if a class type instance is passed, not by reference, but by *value*, as the compiler must arrange for value semantics to be preserved when the procedure is called, by constructing a copy that the procedure can access and use without affecting the original argument.

Here follows a fairly detailed analysis of the relationship of cell to behavior. Suppose in the case of a single cell with a single wire attached to it the cell is an instance of a prototype named "C" having a single *input* port with "p" and "W" as its name and type, respectively. The wire is named "1" while "c" is the name given to the *C* cell instance. Then, omitting for the moment the preamble where classes *C* and *W* are defined, the actual C++ code generated by the Codifier is as follows:

```
BEGIN
     W vv1("1");
BEGIN_CELLS
     C("c", vv1);
END_CELLS
END
```

The omitted preamble also defines the meaning of the bracketing tokens `BEGIN`, `BEGIN_CELLS`, `END_CELLS` and `END`, which are C++ preprocessor **`#define`** "macros" whose definition will be given shortly. For now, suffice it to say that these macros establish the proper context for wire and cell instantiation. The wire is instantiated first, as a stack-allocated automatic variable named `vv1`,[1] an instance of class *W* created by calling its constructor with a string ("1") naming the wire object. The cell named "c" is then instantiated, but with a key difference, syntactically and semantically, which goes beyond the fact that *C*'s constructor takes an additional argument of type *W*. In this case, there is *no automatic variable*, as in the one named `vvc` defined thus:

```
C vvc("c", vv1);
```

The omission is deliberate, and exploits what in C++ parlance is called a *functional cast.* Syntactically identical to a function call, it is normally used for type coercion; that is, "casting" one type as another (possibly more specialized) type, suitable for use in expressions, or passing by value to other functions or procedures.

---

[1]The "vv" is prefixed to the string name "1" by the Codifier, yielding a valid C++ identifier. The purpose of the prefix is twofold. First, it prevents conflicts with any reserved C++ keywords, which are illegal as identifiers. Second, it countenances wire names consisting entirely of numeric characters, which can appear in identifiers anywhere *except* as the first character. The Codifier also filters out of the string name any punctuation or other characters not allowed in identifiers.

This is a feature not available to procedure calls, which, unlike class constructors, cannot return instantiated objects to be used as values. The functional cast results in the compiler constructing and returning a temporary object of type C, whose fleeting existence lasts long enough to affect the `vv1` wire object passed to the constructor, but no longer. After the constructor exits, the C class *destructor* is immediately called. Destructors normally have cleanup responsibilities, such as releasing any storage or other resources an object has acquired during its lifetime. In this case, there is nothing to do, and the transient object quickly disappears. No stack (or heap) variable remains for subsequent use by other cells, which implies that if a cell wants to communicate with another cell, it must do so posthumously through a wire intermediary.

Hence, in Vista protocols, wires are persistent and cells are transitory. To be more precise, wires persist only for the duration of the scope in which they are defined, which is usually *function* scope. The top-level bracketing macros `BEGIN` and `END` delimit this scope, which is the function called *main*, the starting point for all C++ executables. For now, assume they are defined as follows:

```
#define BEGIN main() {
#define END }
```

These are not their real definitions, but will do for this simple example. Later on, when the actual Vista protolibs are described, with working examples, the real definitions will be given. Assume further that the cell scoping macros are defined away, which is perfectly acceptable to the preprocessor and the compiler:

```
#define BEGIN_CELLS
#define END_CELLS
```

Then, after the preprocessor expands these macros (and the vertical "whitespace" is manually removed), the code seen by the compiler is simply:

```
main() {
   W vv1("1");
  C("c", vv1);
}
```

Adding a preamble necessary for this code to be compilable and (minimally) meaningful to Vista shows that would-be wire and cell classes must inherit from the predefined *Wire* and *Cell* base classes:

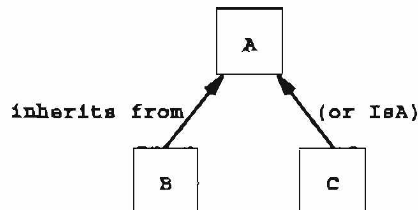
```
class W : public Wire {
public:
  W(const char* name) : Wire("W", name) {}
};
```

```
class C : public Cell {
public:
  C(const char* name, W& wire) : Cell("C", name) {}
};

main() {
   W vv1("1");
  C("c", vv1);
}
```

The protocol for both of these base classes requires their constructors to take an extra string argument in addition to the name string also delegated to them by their subclasses. This extra argument is the name of the subclass *type* as a character string. Its purpose is to provide a rudimentary form of *runtime type identification* (RTTI), the utility of which will be shown below in Subsection 4.3.1.

4.2.1 Type

Now suppose the two cells being connected have ports of different types. As has been stated, the Vista type inference mechanism permits them to be different, so long as they are compatible. The simple inheritance tree below implies the compatibility of types A and B, A and C, and B and C. The wire connecting any of these port pairs would have type A, it being the submost common derived class.



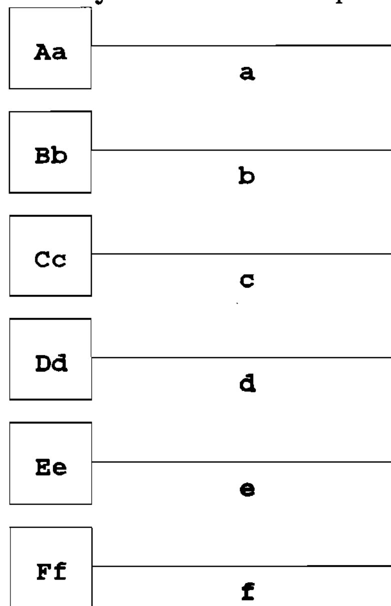
That is, since the "IsA" relation holds, since A is both a B and a C by virtue of (multiple) inheritance, an A can pass for a B or a C.

The following is a somewhat more complex inheritance graph, in the simple textual form Acme sees when reading the file that tells it how to initialize a protolib. Briefly, there is a line for each different type, and for each line the first type listed is the child, and the rest (zero, one or more) are its parents:

```
A
B A
C B
D C
E B
F D E
```

Consider the following six cell instances, one each of types Aa, Bb, Cc, Dd, Ee and Ff, each having a single port of type A, B, C, D, E and F respectively, to which

is attached a single wire named by the lowercase equivalent of the port type name:



If each cell instance is named the same as the wire attached to it, the following *vistafication*[2] (fragment) would result:

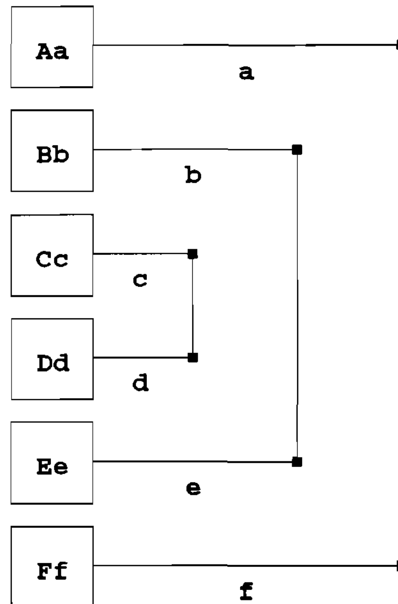
```
BEGIN

    A vva("a");
    B vvb("b");
    C vvc("c");
    D vvd("d");
    E vve("e");
    F vvf("f");

BEGIN_CELLS

    Aa("a", vva);
    Bb("b", vvb);
    Cc("c", vvc);
    Dd("d", vvd);
    Ee("e", vve);
    Ff("f", vvf);
```

[2]This is the third of a trio of convenient neologisms, defined as follows:

vistafy — to specify tool-agent interactions using Vista's framework and protocol.
vistafier — that which vistafies.
vistafication — the process or result of vistafying.

```
END_CELLS

END
```

Connecting the wires, as shown below, makes three wires out of six and changes their types.



Also, when already named wires are merged, Acme renames the new wire by concatenating the names of the old wires, separating them with an underscore character, as shown in the new vistafication below:

```
BEGIN

     F vva_f("a_f");
     D vvc_d("c_d");
     E vvb_e("b_e");

BEGIN_CELLS

    Aa("Aa1", vva_f);
    Bb("Bb1", vvb_e);
    Cc("Cc1", vvc_d);
    Dd("Dd1", vvc_d);
    Ee("Ee1", vvb_e);
    Ff("Ff1", vva_f);

END_CELLS

END
```

The first wire (named "a_f") has the longest pedigree. F is an E, E is a B, B is an A, hence F is an A by heritage. In terms of the criterion mentioned above

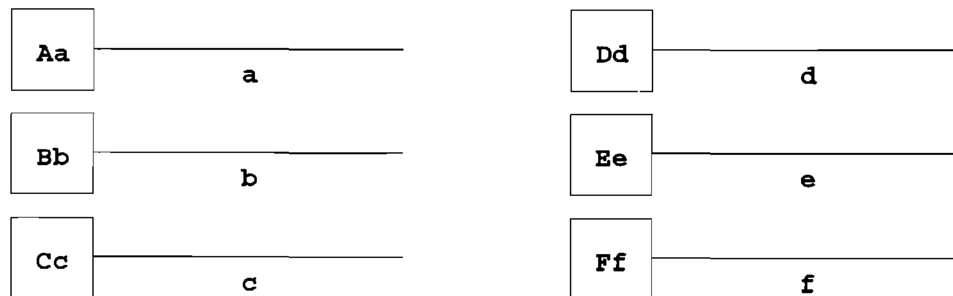
for the simple case, the submost common derived class (or, in the parlance of partially-ordered sets, the *greatest lower bound* or *meet* or *product*) of F and A is F.

4.2.2 Order

Since C++ is a sequential programming language, there is a strict sequence in which cell and wire instantiations must occur. Does it matter in what order wires and cells are instantiated? The Vista protocol insists that all wires be instantiated before any cells, but what about the ordering among the wires, or among the cells? In the six cell-wire pairs vistafication above, the wires are instantiated in the same order as their corresponding cells. In the three wire/six cell case, however, the wire ordering is scrambled a bit. As might be expected, indeed instantiation order *does* matter, but only for cells, not for wires.

In general, the sequence of wire instantiations is neither predictable nor intuitive, as it is based on the traversal order of the wire table maintained by the cellmatrix, since that is how wire instantiations are generated by the Codifier. By contrast, the cellmatrix imposes a well-defined ordering on cell instantiations, which is that of left-to-right, top-to-bottom. This is intuitive, at least for native readers of Western languages who have ingrained in them these major/minor (vertical/horizontal) directions of the printed text.

A minor change in the six cell-wire pairs design demonstrates the difference it makes in the ordering of the vistafication:




```
BEGIN

        D vvd("d");
        C vvc("c");
        A vva("a");
        B vvb("b");
        F vvf("f");
        E vve("e");

BEGIN_CELLS

        Aa("a", vva);
        Dd("d", vvd);
```

```
        Bb("b", vvb);
        Ee("e", vve);
        Cc("c", vvc);
        Ff("f", vvf);

    END_CELLS

    END
```

To recapitulate, wire instantiations are order *in*dependent, while cell instantiations are order *de*pendent. This makes sense, since wires do not interact with other wires unless and until they are acted upon by cells, whose actions, like most data-mutating operations, are non-commutative.

4.3 Practice

With the theoretical foundation properly laid, a presentation of the implementation details of a "real" Vista protolib can proceed. This shift from theory to practice is somewhat slippery, however, as the **sig** protolib is by no means practical. Indeed, as will become obvious, it implements a ridiculously inefficient way to do interprocess communication. Still, it does illustrate what is possible if signals are assumed to be the *only* means of communication at one's disposal.

Exception handling being their primary purpose, Unix signals are generally of extremely limited usefulness as a communication mechanism. Some of the problems with signals are:

- Signals are received asynchronously by the receiving process.
- The only information conveyed by signals is a single integer from a small set representing defined signal types.
- In general, the recipient does not know which process sent the signal.

While used programmatically for many purposes, at the user level signals are normally used to interrupt or terminate running processes at will. This is usually accomplished via the Unix **kill** command issued to the shell by the user who invoked the process. The Unix kernel also sends signals as a kind of software (as opposed to hardware) interrupt, notifying running processes of faulty or exceptional conditions, such as invalid memory access requests. The **signal** system call is used to specify a so-called *handler* for a given type of signal. The handler is a (global) function that will be called immediately upon receipt of the specified signal. When this handler function returns (assuming the signal was not `SIGKILL`, which cannot be caught or handled) the program continues execution exactly where it left off. Hence, signals can only be handled nonhierarchically, completely outside the function call hierarchy of the program. There is no way to tell the program to continue *not* where it left off after catching a signal, but at some other point in the call stack. Short of setting some global state while in the handler function, there is no way for the program to subsequently tell it even caught the signal at all.

Owing to these limitations, using signals for interprocess communication necessitates a cooperative protocol. That is, both sender and receiver must be prepared in advance to deal with an exchange of signals. These preparations include adding the appropriate handler functions and the calls to **signal** to "prime" them. Despite the fact that Vista eschews source code modifications for enabling cooperation, the vistafication described in this section is useful for demonstrating the kinds of modifications that are required, as well as for illustrating basic Vista protocols.

Figure 4.2 shows both the schematic, called `test`, and the vistafied `test.c` file (preamble and all) of yet another reincarnation of the cell-wire-cell triad, where this time the wire represents the Unix kernel and its conduit for signals.

The first line is a comment identifying the name of the file. That plus the first two `#includes` represent the Vista netlist format file header. The other `#includes` represent the format of subcircuit definitions, one for each different type of cell used in the design. The actual definitions, of course, are found in the `.h` header files, whose contents are inserted when the C++ preprocessor does its work.

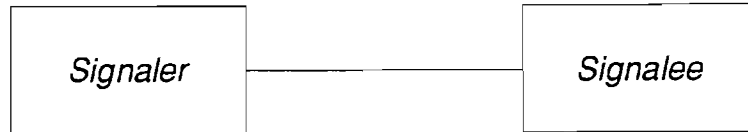
Associated with each Vista design and each Vista protolib is a header file of the same name with a `.h` suffix. The design header is included first, so it can `#define` customizations for the protolib header, which in turn is included before any cell prototype `.h` files. Residing in the protolib header file are the fundamental class definitions and protocols designed for that protolib. The protolib designer *must* provide this file before anything can be vistafied. Initially, the individual design header files *do not* exist, and are created by the Codifier with the following default contents:

```
// File: test.h

#define BEGIN MAIN_BEGIN

#define BEGIN_CELLS MAIN_BEGIN_CELLS

#define END_CELLS MAIN_END_CELLS

#define END MAIN_END
```

The enterprising (human) vistafier[3] is free to include other definitions in this file, by editing it with a text editor. In some protolibs special additions are required by the protocol, but in general, the generated default will do.

The `MAIN_` prefix serves to substitute one set of bracketing tokens for another set, which are defined, not in the `sig.h` file, but in the base *vista* protolib header file `sig.h` `#includes` before doing anything. Included in Appendix A (along with

[3]As used here, vistafier means *one who vistafies.* The appellation applies to any participant in the vistafication process, be it human subject or software object.




```
// File: test.c

#include "test.h" // design
#include "sig.h" // protolib
#include "Signaler.h" // (NAME, MODS, BIDIR(Medium, vvm))
outclude
#include "Signalee.h" // (NAME, MODS, BIDIR(Medium, vvm))
outclude

BEGIN

     Medium vv1("1");

BEGIN_CELLS

    Signaler("self", "bmods=0 Message='ererer'", vv1);
    Signalee("parent", "bmods=1", vv1);

END_CELLS

END
```

Figure 4.2. A Signaler/Signalee Vistafication

the other Vista protolib header files), the `vista.h` file is also where the base *Cell* and *Wire* classes are defined, along with other useful `#define` macros.[4]

The `sig.h` file is presented piecemeal below, interspersed with some explanatory comments. The semicolon following a method's argument list (with `(void)` meaning no arguments) indicates that the method is here declared, but not defined. In fact, the language allows this, and assumes that class function members are defined in a separate `.c` file that includes the `.h` file. In reality, as they are comparatively small (very few lines of code), all Vista methods are *defined when declared* in the header file. In addition, they are ordered so that each method is defined before it is used in the body of another method. This ordering allows the compiler to do efficient, one-pass inlining of all methods.[5]

```
// File: sig.h

#include "vista.h"

class Medium : public Wire {
private:
  int signaler_pid, signalee_pid;
public:
  Medium(const char* name) : Wire("Medium", name) {
    signaler_pid = signalee_pid = 0;
  }
  ~Medium(void);
  Medium& operator << (int pid);
  Medium& operator >> (int& pid);
  }
};
```

The *Medium* type wire is very simple. It has a constructor, a destructor (which does nothing), two overloaded operators, and two integer data members, which are initialized to zero by the constructor. For a valid interaction, the protocol requires the two integers to be assigned the values of the Unix process id of both the *signaler* process and the recipient process, the *signalee*. Exactly how this happens is explained below.

---

[4]Generally disdained by "serious" programmers who consider it bad form to use them for mere syntactic "sugaring" purposes, there are nevertheless good reasons for using `#define` macros. See [103] for detailed justifications.

[5]This will not work in general; for example, one pass will not suffice when two methods call each other.

```
#undef VISTAFIER
#define VISTAFIER sig

class VISTAFIER : public Vistafier {
private:
  int signaler_pid, signalee_pid;
  var message;
public:
  void Init(int argc, char** argv) {
    Vistafier::Init(argc, argv);
  }
  VISTAFIER(void) : signaler_pid(0), signalee_pid(0) {
  }
  ~VISTAFIER(void);
  static void GotSignalAlarm(void);
  int GetSignalerPid(var id);
  int GetSignaleePid(var id);
  int& SignalerPid(void);
  int& SignaleePid(void);
  void GetMessage(var mods);
  void GetMedium(var mods);
  boolean IsMessageReady(void);
  boolean IsMediumReady(void);
  void Send(int sig);

  VISTAFIER& operator << (int bit);
  VISTAFIER& operator << (char c);
  VISTAFIER& operator << (char* s);

  int Exit(void) {
    if (IsMessageReady() && IsMediumReady()) {
      self << message; // deliver message
      return 0;
    } else {
      return 1;
    }
  }
} VISTAFIER;
```

Wherever the VISTAFIER #define preprocessor macro symbol appears it will be replaced with the *sig* identifier. The use of a #define for this substitution is not strictly necessary, but it does emphasize the essential role of this class as the *Vistafier*. Each Vista protolib defines such a class inheriting directly from the base *Vistafier* class defined in the vista.h file. Each *Vistafier* subclass has a single instance (of the same name, in this case *sig*) which is created immediately after the

subclass is defined, by reason of being named between the class closing brace and the semicolon. Thus, as it is declared/defined before `main`, each specialized *Vistafier* class object becomes a global variable accessible to all cell and wire methods.

The global Vistafier class object is the master controller and coordinator of the various vistafication tasks. It plays an important role in the three phases that follow the Codifier phase, as it guides the Analyzer, the Manifester and the Executer in realizing a given tool-agent interaction. In any vistafication, a full interconnection of all cells is transparently effected by the Vistafier.

To illustrate, Figure 4.3 shows the *sig* Vistafier as a "fat" wire interconnecting the signaler and the signalee, and the resulting vistafication were it to be treated as a normal wire. The girth of this Vistafier wire stems from its assimilation of several cell-like actions in support of its service as the key state-carrying object and interaction mediator. In the spirit of global power bus hiding in Acme/PPL, avoiding unnecessary visual and cognitive clutter is the main motivation for making the Vistafier wire *implicitly* as opposed to *explicitly* globally accessible.

In addition to *signaler_pid* and *signalee_pid* (the same two integer data members as in the *Medium* wire class), *sig* maintains a *message* member of type *var*. Short for *variable*, *var* is used throughout Vista as a multipurpose data object that can either be staticly typed, or else can assume its type *at runtime*, based on the context of its use. Subsuming such basic types as *int*, *long*, *float*, *double*, *char** or even *string* into one "super-string" class, the *var* class, via operator overloading, implements arithmetic, string and sub-string operations, mixed operations between mixed types, and formatted stream output. With a *VarMap* class providing associative arrays of *vars* indexed by *vars*, this superbly useful C++ library serves Vista very well.

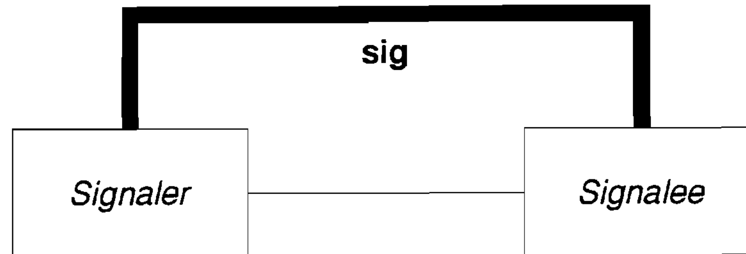
The *Init* method of *sig* is listed before the constructor, not only because the constructor calls it, but because this method is what really breathes life into a staticly-constructed vistafier. *Init* extracts command-line arguments from the execution environment as soon as they become available, as seen to by the `MAIN_BEGIN` macro, which is shown expanded below. With no member maintained directly by *sig* needing initialization, its *Init* method merely delegates to the *Init* method of its superior *Vistafier* class.

The purpose of the other declared *sig* methods and the three overloaded << operators will be revealed shortly. The *GotSignalAlarm* member function is special, by virtue of being declared *static*,[6] which makes it suitable as a handler function passed to **signal**, as shown in Subsection 4.3.3.

The *Exit* method shown above in its entirety is the counterpart of the *Init* method. This method is the focal point for the signal-based interaction of this Vista protolib. The cells and the *Medium* wire only prime the pump, so to speak. Before showing the protocol of the *Signaler* and *Signalee* cells, and the definitions of the *Medium* << insertion operators, the heart of this test vistafication is first laid bare, with all `VISTAFIER` and begin/end bracketing macros fully expanded:

---

[6]Static member functions are like global function, but their access is more controlled, and they do not pollute the global namespace, as their names are scoped within the class. Hence, outside of class methods, C++ static member functions must be referenced using the `::` scope resolution operator, e.g., `sig::GotSignalAlarm`.




```
class Medium : public Wire {
  // ...
};

class sig : public Vistafier {
  // ...
} sig;

BEGIN

     Medium vv1("1");

BEGIN_CELLS

    Signaler("self", ".....", vv1, sig);
    Signalee("parent", "...", vv1, sig);

END_CELLS

END
```

Figure 4.3. A Signaler/Signalee Vistafication with Explicit Vistafier

```
int main(int argc, char** argv) {
  sig.Init(argc, argv);
     Medium vv1("1");
  do {
    Signaler("self", "bmods=0 Message='ererer'", vv1);
    Signalee("parent", "bmods=1", vv1);
  } while (sig.IsVistafying());
  return sig.Exit();
}
```

Whatever it is they do, the two cells apparently require more than one lifetime to do it, as they are instantiated anew each time around the do/while loop controlled by the *IsVistafying* method. This is a virtual base class method, and may thus be overridden in subclasses of *Vistafier*, although *sig* has no reason to do so, as the default base behavior is adequate (see the technical report). The controlling hand of the Vistafier as Analyzer/Manifester/Executer is what distinguishes each reincarnation of these two cells, whose time for the limelight has come:

```
// File: Signaler.h

defCell(Signaler, (NAME, MODS, BIDIR(Medium, m)))
  if (VISTAFIER.IsAnalyzing()) {
    BPORT(name, m);
    m << VISTAFIER.GetSignalerPid(name);
  }
  else if (VISTAFIER.IsManifesting()) {
    m >> VISTAFIER.SignaleePid();
  }
  else if (VISTAFIER.IsExecuting())
    VISTAFIER.GetMessage(mods);

// File: Signalee.h

defCell(Signalee, (NAME, MODS, BIDIR(Medium, m)))
  if (VISTAFIER.IsAnalyzing()) {
    BPORT(name, m);
    m << VISTAFIER.GetSignaleePid(name);
  }
  else if (VISTAFIER.IsManifesting()) {
    m >> VISTAFIER.SignalerPid();
  }
  else if (VISTAFIER.IsExecuting())
    VISTAFIER.GetMedium(mods);
```

The `defCell` macro, together with `NAME`, `MODS` and the three port mode macros

INPUT, OUTPUT and BIPUT (or BIDIR), are convenient shorthand for declaring a *Cell* subclass with a constructor that forwards both name and type to the base *Cell* class constructor. The default skeleton generated by the Codifier when no .h file yet exists is the defCell macro call with no body,[7] which when expanded for *Signaler* (similarly for *Signalee*) looks like:

```
class Signaler : public Cell {
public:
  Signaler(const char* name, var mods, Medium& m)
    : Cell("Signaler", name) {
```

During their first life, that is, while the *sig* vistafier *IsAnalyzing*, the *Signaler* and *Signalee* both have two similar actions to perform on their shared wire, shown here with macros expanded for the former:

```
m.Port('b', name, "m");
m << sig.GetSignalerPid(name);
```

The purpose of the *Port* wire method is to record two facts in the test.a log file mentioned in Subsection 3.4.4, those facts being that there is a port on the cell and a *terminal* on the wire, "m" naming them both. (Actually, since there is usually more than one port with the same name, the name "m" is qualified (prefixed) with the instance name of the cell it belongs to.) The reason for distinguishing a port from a terminal, the former being cellside, the latter wireside, is to make it easier to write connectivity rules for the Manifester, shown below in Subsection 4.3.1.

The *sig* methods *GetSignalerPid* and *GetSignaleePid* are called with the cell name as an argument.[8] Note that it is no accident that the Signaler and the Signalee were named "self" and "parent" respectively, as these names control the lookup of the appropriate process id:

```
int GetSignalerPid(var id) {
  if (id == "self") {
    return getpid();
  } else {
    return 0;
  }
}
```

---

[7]Note that the outclude macro in test.c supplies the brace closing the constructor definition, as well as the final brace and semicolon closing the class definition.

[8]In another appearance of the *var* type, these methods take a *var* argument, using both the automatic type conversion (const char* to *var*) and the overloaded equality operator (which does a string comparison) defined in the *var* class.

```
int GetSignaleePid(var id) {
  if (id == "parent") {
    return getppid();
  } else {
    return 0;
  }
}
```

The Unix system calls **getpid** and **getppid**, return the id of the currently running process and its parent process, respectively. If either cell name does not conform, zero is returned, and the *sig* protocol fails. Note that the "else" branch of the *GetSignaleePid* "if" could return, instead of zero, the id of an arbitrary process (not named "parent") by an operating system query finding the id given the process name.

Now known, the process ids of the Signaler and the Signalee are "pushed" into the *Medium* wire, via its << operator:

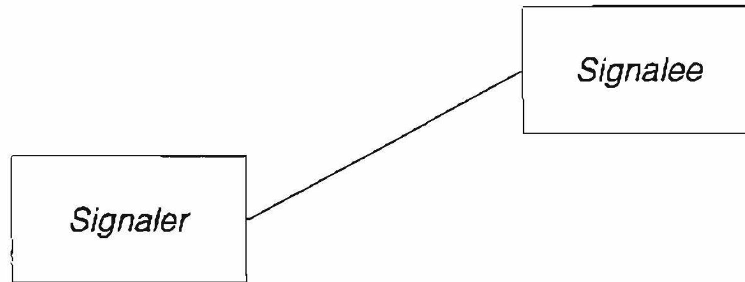
```
Medium& operator << (int pid) {
  if (signaler_pid == 0) {
    signaler_pid = pid;
  } else {
    signalee_pid = pid;
  }
  return self;
}
```

Stored there only temporarily, they await being "pulled" from the wire when, in the second act, the *sig* vistafier *IsManifesting*. The >> operator does the pulling:

```
Medium& operator >> (int& pid) {
  if (signalee_pid != 0) {
    pid = signalee_pid;
    signalee_pid = 0;
  } else {
    pid = signaler_pid;
    signaler_pid = 0;
  }
  return self;
}
```

After two pushes and two pulls on this wire, its *signaler_pid* and *signalee_pid* members will be zero once more, as they were in the beginning. The two ids are put into the corresponding members of the *sig* object. The purpose of this convoluted two-phase data exchange is to ensure the validity of a *sig* vistafication, making it impossible to carry out a signal-based interaction unless the cells are

hooked up correctly. As shown below in Subsection 4.3.1, the Manifester will catch an invalid configuration and explain its invalidity, but since the Manifester stage can be bypassed, it is prudent to have a separate, backup validation scheme. Also, this scheme is indifferent to cell instantiation order, so if the user were to place the cells so that the *Signalee* were instantiated first, as below, it would still work:



4.3.1 Making Manifest

During the transition from its Manifesting state to its Executing state, the *Vistafier* method *MakeManifest* is called, which uses string concatenation and type conversion operations defined on *var*:

```
var makecmd("make");
makecmd += " manifest OUT=";
makecmd += basename;
status = system((char*) makecmd);
if (status > 0) state = DONE;
```

Since `test` is the "basename" for this vistafication, the `char*` string passed to **system** would be:

```
make manifest OUT=test
```

If anything goes wrong, and **make** returns an error status, the state variable stored in the *Vistafier* class will be assigned the (enumerated-type) value `DONE`, and a value of *false* will be returned from the *IsExecuting* method, preventing any further execution of this vistafication. If everything goes right, and **make** succeeds, it will have done the following five steps:

1. massage the output of the Analyzer into a `test.F` facts file,
2. combine that file with the files `sig.P` and `sig.q` which contain, respectively, the rules and the query associated with the *sig* protocol,
3. fuse it all into one `test.P` module file,
4. create a `test.q` command file that refers to the `test.P` module, and

5. invoke **coral** (the logic programming system) with `test.q` as input.

The `test.F` file contains seven facts, one for the wire and three for each of the two cells, showing why cells and wires need to know their type (see Subsection 4.2.1):

```
/* File: test.F */
medium("1").
signalee("parent").
bport("parent_m","parent").
terminal("parent_m","1").
signaler("self").
bport("self_m","self").
terminal("self_m","1").
```

In `sig.P` are found the following eleven rules, where, following standard Prolog notational conventions, the uppercase terms (C, W, CR, etc.) are variables, the lowercase terms are predicates, and the `:-` connective separates the head of a rule from its body. The mnemonics C for Cell, W for Wire, NM for Number of Mediums, etc., are pretty much self-evident.

```
 1. cell(C) :- signaler(C).

 2. cell(C) :- signalee(C).

 3. wire(W) :- medium(W).

 4. port(P,C) :- iport(P,C).

 5. port(P,C) :- oport(P,C).

 6. port(P,C) :- bport(P,C).

 7. connected(C,W) :- port(P,C), terminal(P,W).

 8. numsignalers(count(<R>)) :- signaler(R).

 9. numsignalees(count(<E>)) :- signalee(E).

10. nummediums(count(<M>)) :- medium(M).

11. ok(CR,CE,WM,NR,NE,NM) :- connected(CR,WM), connected(CE,WM),
                             numsignalers(NR), NR = 1,
                             numsignalees(NE), NE = 1,
                             nummediums(NM), NM = 1.
```

The first six rules express "IsA" inheritance relationships, with a general predicate in the head of the rule and a specific predicate in the body of the rule. That is, the first rule simply says that C is a cell if C is a signaler. Likewise, the sixth rule says that P is a port of C if P is a bport of C.

The seventh rule is for connectivity. Stated in English, it says that C is connected to W if P is a port of C and P is a terminal of W. Rules eight, nine and ten illustrate usage of the built-in **coral** *multiset* operators. The *grouping* operator < ... > is one way to create a multiset, the cardinality of which is returned by the *count* operator. Thus, informally, rule ten says that if M is a medium, then include it in the set of all mediums, and count the total members of the set.

Rule number eleven puts it all together, using an *infix* equality (=) predicate to say that everything is okay if the same wire is connected to two different cells, and there is exactly one signaler cell, one signalee cell, and one medium wire.

The five commands and lone query given to **coral** in the `test.q` input file are the following:

```
consult(test.P).
explain_on.
?ok(CR,CE,WM,NR,NE,NM).
explain_off.
shell("Explain -f dump_directory").
quit.
```

The *consult* command tells **coral** to read and evaluate the `test.P` module containing the facts and rules just shown. The query is formed by tacking a question mark onto the head of rule eleven. Bracketing the query by the *explain_on* and *explain_off* commands results in the creation of a subdirectory called `dump_directory` containing a single file (`ok.d`) where **coral** "dumps" a record of each rule instantiation that generates a fact. Just before **coral** quits, the *shell* command (via **system**) fires up the **Explain** program, telling it look for all dump files in the dump directory. Mentioned in Subsection 3.4.3, **Explain** is the facility that knows how to read the `ok.d` dump file, and display it in tree form.

Figure 4.4 shows the main window that appears when **Explain** starts up. In the file browser are listed all currently loaded dump files, which in this test case, is the lone `ok.d` file. In general, each different query being explained will have its own file and entry in the browser.

Selecting the solitary entry and choosing the "Show Selected Modules" command in the "Modules" menu will "consult" the dump file, and then bring up the derivation window displayed in Figure 4.5, where the user may dynamically "grow" or "prune" the derivation tree by clicking the mouse on its leaves, and thus explore all derivations produced by the query. This tree provides a visual explanation of the iterative process of fact generation by rule instantiation.

The user must first click on a predicate name in the top-left "Predicates" browser, whereupon all of the instantiations for (facts with) that predicate will appear in the bottom-left "Instantiations" browser. The user then clicks on an instantiation, causing it to be rendered in tree form in the main drawing window. The root of the

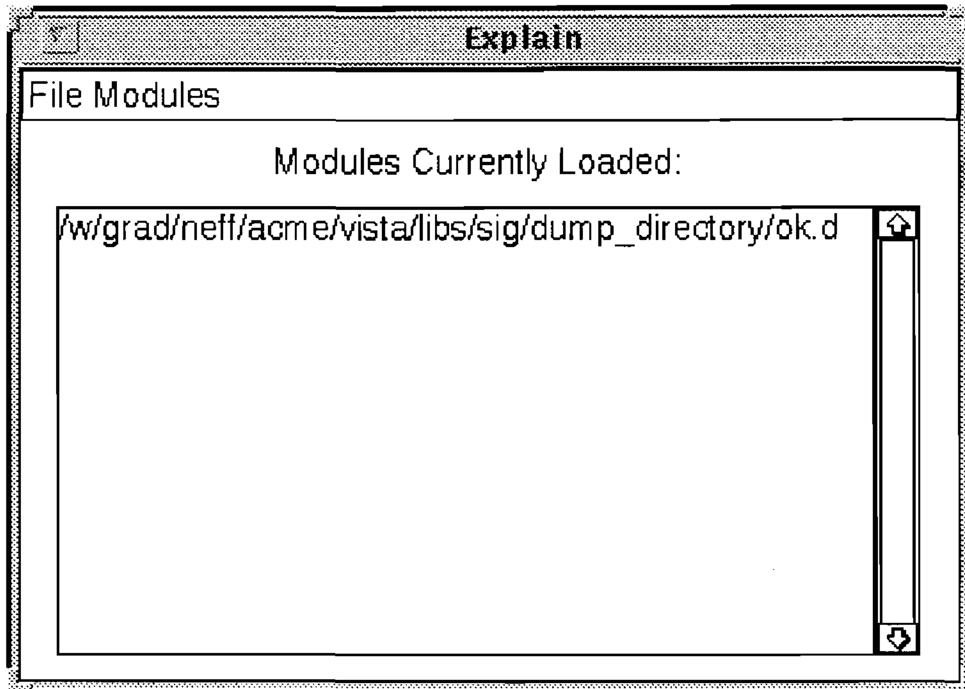


Figure 4.4. The **Explain** Main Window

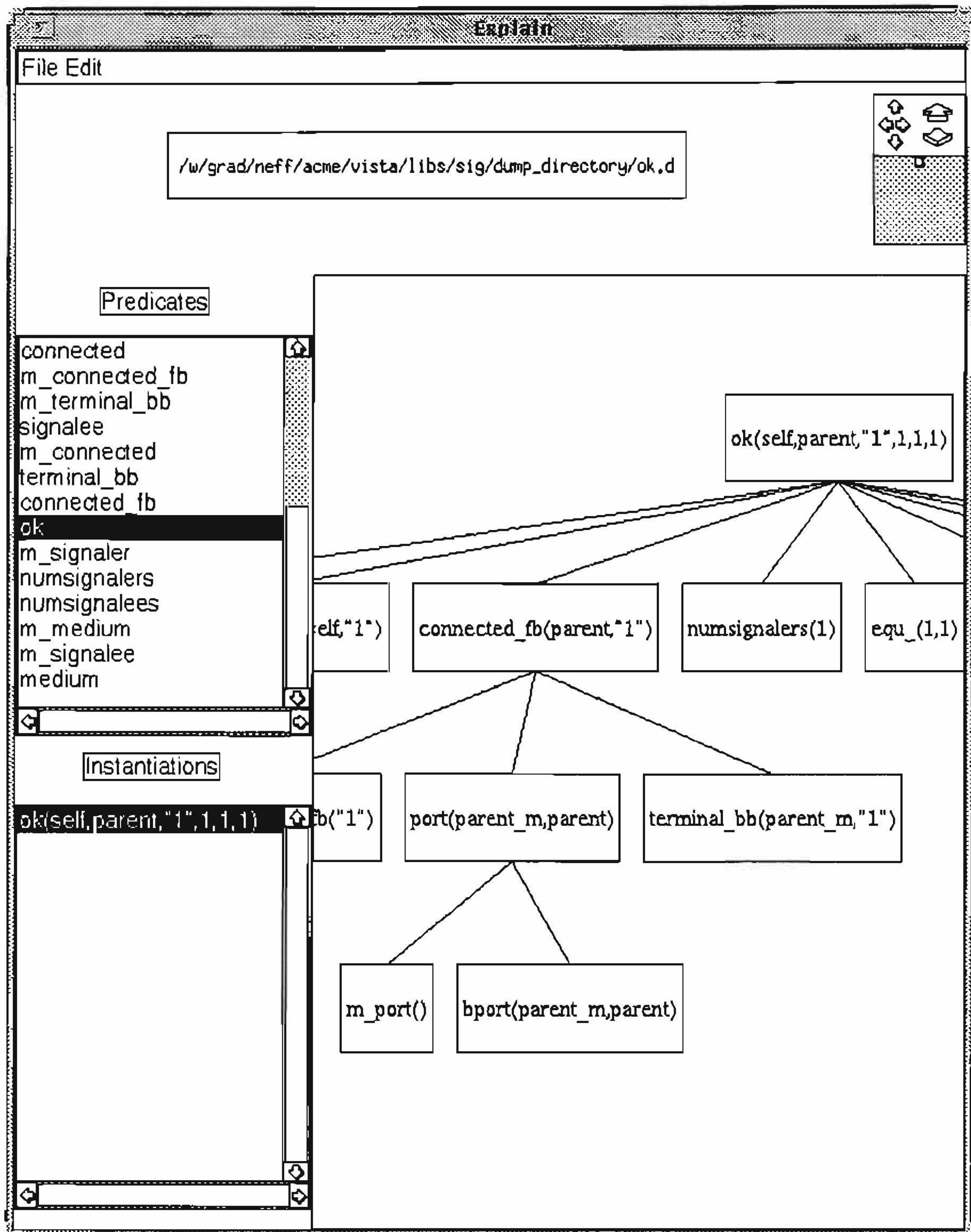


Figure 4.5. The **Explain** Derivation Window

tree represents the derived fact, while the children springing from the root represent the facts in the body of the rule.

The tree in Figure 4.5 is the one that appears when the single instantiation of the "ok" predicate is selected, and two leaves are expanded. The extraneous nodes where the predicate names are prefixed with "m_" are due to the "magic" rewriting **coral** does to "materialize" evaluation of the `test.P` program. Note too that some predicate names have suffixes like "_fb" indicating which arguments were *free* and which were *bound* in the evaluation of a subgoal, such as `?connected(X,"1")`. The fact `connected_fb(parent,"1")` indicates that `connected(parent,"1")` was computed in response to the query fact `m_connected_fb("1")`. See [5] and [76].

Closing the **Explain** windows terminates the program and returns control in rapid succession to **coral**, **make** and the test vistafier, which then completes the transition to its Executing state, triggering the third life of the signaler and the signalee. Their respective roles in this final state are to communicate to the vistafier the message to be sent, and the readiness of the medium. This they do by passing their "mods" (short for *modifiers*) parameter to the *Vistafier* methods *GetMessage* and *GetMedium*.

4.3.2 Parameterization

The *mods* string, which is coerced into a *var* type on cell instantiation, currently ranks next to the instance name as the easiest and most flexible means of parameterizing[9] a cell. (In Chapter 6, more elaborate mechanisms for cell parameterization will be explored.)

In Acme, a *modifier* is a local state object associated with a cell instance. In a language analogy, if a cell is a verb then a modifier is an adverb. Each cell prototype can have any number of modifiers associated with it, which the user may define for cellmatrix prototypes when creating them via the *Wrap* command. Then, using the *Modify Cells* command, the user can set different values in these modifier objects for different instances of the same prototype.

Acme currently has three types of modifiers—*BooleanModifier*, *StringModifier* and *GroupModifier*, all of which inherit from the base *Modifier* class. Boolean modifiers are used to assert/deassert named predicates (e.g., *HasLoad*) to tell whether or not a given cell instance should have some modification, either physical, structural or behavioral (e.g., a load placed on an internal node). String modifiers provide a means of attaching named attributes to cell instances in the familiar form of a property list of name/value pairs. Group modifiers allow arbitrary combinations of boolean and string modifiers to be grouped in named hierarchies.

For each Acme cell with modifiers defined on it, the Codifier builds a stylized string of modifier names and values for automatic inclusion as the second parameter of Vista cell constructors. All boolean modifier values are assembled bit by bit into

---

[9]Parameterization has long been used to solve the flat namespace pollution problem caused when every minor variation of a general theme is named slightly differently. For example, a series of prototypes named 1BitALU, 2BitALU, 4BitALU and so forth, is both undesirable and unnecessary if instead a single prototype, NBitALU, can have a parameter named N, the bit-width. Parameters partition the namespace hierarchically, showing the essentials while hiding the accidentals.

a single integer, which is inserted as the value of the "bmods" name. That is, the "bmods" string, an equals sign and this integer are concatenated to form the first token in the *mods* string. String modifiers come next, but as their values may contain spaces, delimiting characters (such as the ' (single-quote)) are needed to surround the values following the modifier name and the equals sign.

The "parent" Signalee has a single boolean modifier named *CanRespond*, which when asserted has a value of 1, and so its *mods* string is:

```
"bmods=1"
```

The "self" Signaler has a string modifier named *Message*, and no boolean modifiers, hence the 0 following the first equals sign in:

```
"bmods=0 Message='ererer'"
```

In this scheme, the "some modifiers defined, none of them boolean" case is indistinguishable from the "boolean modifiers defined, but none asserted" case. However, the user who creates a prototype with defined modifiers is presumably the one who provides the contents of the corresponding `.h` file, and knows which is the case. In the following chapter an example will be shown of a cell with several boolean modifiers, for which this scheme provides the benefit of compactness in the *mods* string traded off against the cost of extracting the bit values from it.

4.3.3 Cooperation

The message the Signaler wants to send to the Signalee is the value of the *Message* string modifier, namely:

```
ererer
```

The meaning of this message will be explained shortly. The successful extraction and caching of this message by *GetMessage* will result in true being returned when *IsMessageReady* is called. Similarly, if *GetMedium* extracts the boolean value "1" from the Signalee's modifiers, then *IsMediumReady* returns true when called. Once it ascertains that the signalee can respond, a necessary side effect of *GetMedium* is to set up the handler function for the `SIGALRM` response that will be sent by the signalee process:

```
signal(SIGALRM, sig::GotSignalAlarm);
```

Repeating for easier reference the *sig::Exit* method, only this time with the `self` macro revealed as applying the `*` indirection operator to the `this` pointer, yielding the `sig` object pointed at, it says that if the message is ready and the medium is ready, then deliver the message and return 0 (meaning success to Unix), otherwise indicate failure by returning 1:

```
int Exit(void) {
  if (IsMessageReady() && IsMediumReady()) {
    (*this) << message; // deliver message
    return 0;
  } else {
    return 1;
  }
}
```

The details of the three << operators are found in the technical report. Briefly, the operator taking a string calls the one taking a character, which in turn calls the operator taking an integer argument, which in turn calls the *Send* method. Thus, the message string is delivered character by character, each character in turn being delivered bit by bit (eight in all) by passing *Send* the value of user-defined signals `SIGUSR1` or `SIGUSR2`, according as the bit value is 0 or 1. It is in the *Send* method that the rubber meets the road:

```
void Send(int sig) const {
  kill(signalee_pid, sig);
  if (signaler_pid > 0) {
    pause();
  }
}
```

The *kill* function is the Unix system call counterpart to the **kill** command. It is what actually delivers the argument signal to the signalee process by way of the Unix kernel. Since *Send* could be invoked to send any signal to any process, the test for a *signaler_pid* greater than zero is included, so that a zero value can indicate that the recipient process is *not* prepared to respond, bypassing the Unix *pause* system call that otherwise will put the signaling process to sleep, awaiting acknowledgement from the signalee. If coming, this acknowledgement must be in the form of a `SIGALRM` signal, which will be "caught" by the do-nothing *GotSignalAlarm* handler, with no other effect than awaking the sleeping signaler causing *pause* (and therefore *Send*) to return, and execution to continue.

For this handshake protocol to work, the cooperation required of Acme consists of the addition of a mere handful of code, starting with two calls to *signal* to set up handlers for the off/on bit signals:

```
signal(SIGUSR1, handle_signal_usr1);
signal(SIGUSR2, handle_signal_usr2);
```

These handlers are two-line global functions that pass 0 or 1 respectively to the *interact_with_char_recognizer* global function, and then call the one-line global function *acknowledge_signal*:

```
void handle_signal_usr1(void) {
  interact_with_char_recognizer(0);
  acknowledge_signal();
}

void handle_signal_usr2(void) {
  interact_with_char_recognizer(1);
  acknowledge_signal();
}

void acknowledge_signal(void) {
  kill(acme()->GetAckPid(), SIGALRM);
}
```

The last thing done by the Executer is the caching by Acme of the id of the just-forked child process, which for this test vistafication *is* the signaler process, Acme being the signalee. Given that the global function *acme* returns a pointer to the sole instance of a class whose *GetAckPid* method returns this cached id, *acknowledge_signal* delivers the necessary `SIGALRM` to wake up the sleeping signaler process.

The bulk of the work is done by the *interact_with_char_recognizer* function, in concert with a staticly-instantiated *char_recognizer* object. In a simple demonstration of code reuse through inheritance, the *CharRecognizer* class is derived from the *FlaggedObject* class, which is a utility class used extensively in Acme. The *FlaggedObject* class provides 32 named (enumerated) boolean flags packed (in most architectures) into the 32 bits of an integer member of the class, with methods to *Set* or *Reset* a given flag, *Clear* all flags, and return the *Bits* of the packed integer.

```
class CharRecognizer : public FlaggedObject {
private:
  enum CharRecognizerState
       { B0, B1, B2, B3, B4, B5, B6, B7 } state;
public:
  CharRecognizer(void) {
    state = B0; flags.Clear();
  }
  void DoBit(int bit) {
    flag_name_t fnt = flag_name_t(state);
    if (bit == 0) flags.ResetFlag(fnt); else
    if (bit == 1) flags.SetFlag(fnt);
  }
  boolean GotChar(int b) {
    switch (state) {
      case B0: DoBit(b); state = B1; return false;
      case B1: DoBit(b); state = B2; return false;
```

```
      case B2: DoBit(b); state = B3; return false;
      case B3: DoBit(b); state = B4; return false;
      case B4: DoBit(b); state = B5; return false;
      case B5: DoBit(b); state = B6; return false;
      case B6: DoBit(b); state = B7; return false;
      case B7: DoBit(b); state = B0; return true;
      default: return false;
    }
  }
  unsigned char GetChar(void) {
    return (unsigned char)flags.Bits();
  }
} char_recognizer;

void interact_with_char_recognizer(int bit) {
  if (char_recognizer.GotChar(bit)) {
    unsigned char c = char_recognizer.GetChar();
    acme()->HandleKey(c);
    acme()->Flush();
  }
}
```

The *GotChar* method implements the straightforward transitions of this recognizer through its eight enumerated states, one for each bit of the character being recognized. Only after a complete cycle through these eight states does this method return true, whereupon *GetChar* is called to return the character *GotChar* just built bit by bit. The Acme *HandleKey* method is then given this character, which maps it to the associated command that is summarily invoked, after which the Acme *Flush* method ensures that any feedback output generated by the command is immediately flushed.

Thus, in the message of this test example, the 'e' character maps to the *Enlarge* command, and the 'r' character maps to the *Reduce* command in Acme. So, if the message is received correctly by Acme, the result is that the view is first enlarged and then reduced back to its original size. This "zoom in then back out" action is repeated three times in rapid succession, producing a simple animation effect.

4.3.4 Caution

The first time this protocol was tried, it did not work. The reason stems from the asynchronous nature of signal receipt, and the race condition caused by the interplay of the Unix signal generator and the Unix scheduler. As the scheduler controls which process is currently running, and how long that process can run before another process is allowed to run, what the scheduler apparently did was to *suspend* the signaler process right after the *kill* call in *Send*, and *before it could pause*. Then Acme, the signalee process, was given a CPU time-slice long enough to receive the `SIGUSRn` signal and reply with `SIGALRM`, which signal was delivered

by Unix to the signaler the instant the scheduler started it running again. The signaler stopped to handle this SIGALRM and then continued executing where it left off, which was right before the call to *pause*. Then it paused, having already consumed the signal that was meant to unpause it, thereby losing its chance to continue with the bit by bit signal/reply handshake.

Because of this unpredictable race condition, Acme must wait long enough for the signaler process to pause before sending its SIGALRM. By experiment, 100000 microseconds (one-tenth of a second) was found to be sufficient when the operating system was lightly loaded, hence the revision:

```
int sleeptime = 100000;

void acknowledge_signal(void) {
  usleep(sleeptime);
  kill(acme()->GetAckPid(), SIGALRM);
}
```

However, this gives no iron-clad guarantee that it would still work under heavily-loaded conditions. Increasing the Acme sleeptime heightens the probability that it would work, but also slows down the message delivery substantially.

The main lesson learned here is how fraught with peril is any attempt to rigorously prove the correctness of interaction protocols involving asynchrony and the Unix operating system, whose formal semantics are chimerical at worst, and highly complex at best. Vista protolibs supporting synchronous protocols fare much better, but still blanch in the face of Dame Formal.

4.4 Pushing and Pulling

Figure 4.2 can be viewed as two cell-wire pairs joined wire to wire, the two wires becoming one. As shown in the previous chapter, this unification can only happen if the two wires are type-compatible, which makes the merger meaningful. Then it makes sense for the two cells to interact by the medium of their common wire. One possible and very common protocol is for the first cell to write data to the wire that the second cell then reads, whereupon the second cell replies by writing data to the wire for the first cell to read. This simple round-trip message exchange, or handshake, is viewed as two cells interacting by mutual acting on a shared wire object. The handshake protocol dictates the proper sequence of actions.

Many other protocols are possible, but be they unidirectional notification-based, or client-server transaction-based, or whatever—the common denominator is always the wire, and the process of *changing its state* through cell-encapsulated actions. These state-changing actions can be viewed, in a fundamental sense, as a *push* or a *pull* on a wire by a cell. While the wire can also be treated as the *indirect* object; that is, a cell pushes *something* onto (or into) a wire, or pulls *something* therefrom, the concept of a wire being pushed or pulled through its state space by force-like cells seems a more parsimonious viewpoint.

The operators $<<$ and $>>$ provide a suitable notation for expressing the respective actions (forces, effects) of pushing and pulling. The labeling of a cell with one

of these symbols is adequate to depict that the sole function of the cell is to invoke a defined << or >> operator on the wire attached to the cell, as follows:

```
defCell(Pusher, (NAME, OUTPUT(WireTypeA, a), INPUT(Source, pushee)))
  a << pushee;

defCell(Puller, (NAME, INPUT(WireTypeB, b), OUTPUT(Sink, pullee)))
  b >> pullee;
```

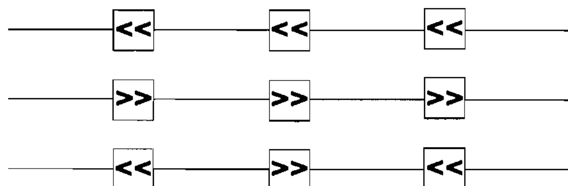
As shown, by using additional ports the source or sink for the data pushed to or pulled from these wires can be passed as a parameter. Taking a cue from the C++ *iostream* class library, which defines several I/O *manipulators* that modify some piece of state maintained by the base *istream* and *ostream* classes, and assuming all wire types are derived from these base classes, the following completely general manipulations are possible (given the existence of a pair of cellname-parameterized functions returning the desired stream manipulators):

```
defCell(Pusher, (NAME, OUTPUT(ostream, os)))
  os << get_ostream_manipulator_from_name(name);

defCell(Puller, (NAME, INPUT(istream, is)))
  is >> get_istream_manipulator_from_name(name);
```

Although not used in the wire class of Section 4.3 above, the return value of its overloaded << operators is the *Medium* wire object itself. This idiom is borrowed from the *iostream* library, which overloads the << (insertion, or "put to") and >> (extraction, or "get from") operators, and allows their convenient juxtaposition:

```
some_type_of_ostream << source1 << source2 << source3;
some_type_of_istream >> sink1 >> sink2 >> sink3;
some_type_of_iostream << source1 >> sink1 << source2;
```

The third case shows how these two operators can be intermixed for bidirectional streams. It would be helpful were this notational convenience able to carry over to the visual Vista language, as in:



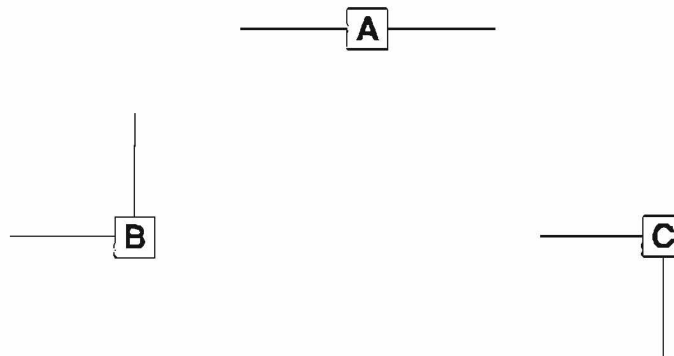
In fact, Vista *can* express such manipulation chains by means of Acme's *equivalent ports* mechanism. Equivalent ports are two or more spatially distinct ports that

have the same name and type, *or* are directly connected *inside* the cell. The netlist generation code computes equivalent ports, and emits in subcircuit definitions and calls only one port/wire parameter for each set of equivalent ports it finds. This is necessary, because in most (if not all) languages the inclusion in a procedure definition of more than one parameter with the same name is an egregious error.

The possibility of having equivalent ports complicates vistafication interpretation. On its face, the following looks like two wires and an intervening cell:

A look inside the cell (recall from Section 3.3 the operation of *clear wrapping*) may reveal the following:

The wire inside the cell connects the two opposite ports, hence, when vistafied only one wire will be instantiated and passed to the lone cell, whose constructor will have been defined to take a single wire parameter. The vistafication is logically isomorphic to that of a cell-wire pair. Thus, in designing cells, even simple attention to the decision of where to put the ports can pay some positive dividends. For example, that the two wires are actually one and the same seems intuitively more likely in cell A than in either cell B or cell C:



In any case, the appropriate use of text labels for disambiguation is well warranted, e.g.:

Of course, the ultimate arbiter must be the cells' textual definitions, including clarifying comments:

```
defCell(IntPusher, (NAME, OUTPUT(ostream, os), INPUT(int, i)))
  os << i; // invokes operator << (int) on os

defCell(FloatPuller, (NAME, INPUT(istream, is), OUTPUT(float, f)))
  is >> f; // invokes operator >> (float) on is
```

4.5 Finite State Machines

The Vista solution to the interobject protocol specification problem discussed earlier in Section 4.1 is embodied in its **fsm** (Finite State Machine) protolib, wherein cells map not only to actions, but to *states* as well, with wires representing *transitions* between states. This mapping, derived from Martin[59] and described below, further demonstrates the dualistic versatility of Vista entity/relationship specifications. In this case, what is flowing through the wires is not data, but rather the *locus of control* of the computation.

Implementing a Finite State Machine (FSM) base class with the idea of subclassing it to realize specific FSM behavior is innately problematic. One of the problems can be seen by looking at the way an FSM was implemented in the *CharRecognizer* class shown above in Subsection 4.3.3, where the states are delineated by a class-private enumerated type, which cannot be extended by inheritance. Outlined in [59], Martin's elegant solution is to decompose an FSM into *two* components, only one of which is subclassed. This method is briefly described below.

FSMs are inherently event-driven. When an event occurs, the FSM undergoes a transition to a new state. A behavior or set of behaviors is associated with each transition. In Martin's scheme, all of these behaviors are bundled into a class, called the *context*, which can be derived from to extend the functionality of the FSM.

All of the state-to-state sequencing happens in the other part of the FSM, the *control* component, which is implemented as a single inheritance hierarchy. Each different state is defined as a class derived from a common base state class, and overrides all the virtual functions defined in this common class. These overriden virtual functions, one for each different event, contain very little code, as they represent only the "wiring" that connects the context behaviors, controlling their execution sequence. This separation of FSM context and control makes new FSMs meaningfully derivable from old ones. The context can thus be furnished with new functionality, and then, for further flexibility and extensibility, can even be given a new control component.

The following five figures present an example of moderate complexity: modeling a search task in a word processor. Figure 4.6 shows a Vista **fsm** design in the form of a state transition diagram[10] from which the Codifier produces the vistafication shown in Figure 4.7, which when *executed* generates the C++ code shown in Figure 4.8 and Figure 4.9.

The context class for this word finder FSM is not automatically generated, as this is where the real functionality resides. Figure 4.10 sketches this class, omitting all but a few details. The following chapter contains a complete and fully-detailed example of a more complex recognizer FSM, used to recognize and strip comments from C++ source code (see Section 5.4 on page 115).

4.6 Capitulation and Recapitulation

A plethora of articles, books, programs and systems have been written for the purposes of promoting, teaching, enabling and facilitating the use of object-oriented programming, and, more recently, the related disciplines of object-oriented analysis and object-oriented design. For example, [29] presents in tutorial style guidelines for using types and inheritance in OOP, such as, *Don't use subtyping for objects that are not the sum of their parts*. Representative of *style* guides is [50], while a much more ambitious guide is the *Fusion*[16] methodology, a very recent CASE approach addressing the many problems associated with object classification and modeling, and behavior partitioning. Vista lays no claim to being one of the premiere "Guides for the Perplexed" in these areas. Although tool-agent interactions can be as general and fine-grained as one likes, depending on one's definitions of tool and agent, dealing with real, non-cooperative but useful tools is still Vista's main concern, and hence its handicap.

In this chapter, the protocols and pragmatics of Vista have been expounded. As has been argued, because objects are more a *binding* mechanism than a *modeling* mechanism, Vista clings to the fundamental yet slippery dichotomy of object and behavior. Targeting C++, Vista merges declarations and definitions, using explicit delegation to superior, inferior or peer objects to execute specifications. From Analyzing to Manifesting to Executing, the omnipresent Vistafier object manages the interaction of cells and wires as they follow the dictates of a given protolib's protocol.

All in all, Vista protocols have as yet only attained plausibility. The demonstrability of the versatility of their applicability is now both imperative and imminent.

---

[10] Observe that there is nothing sacrosanct about squares or rectangles for cell shapes—any two-dimensional shape is permissible. In this case, the chosen shapes accentuate the distinction between states (circles) and actions (diamonds).

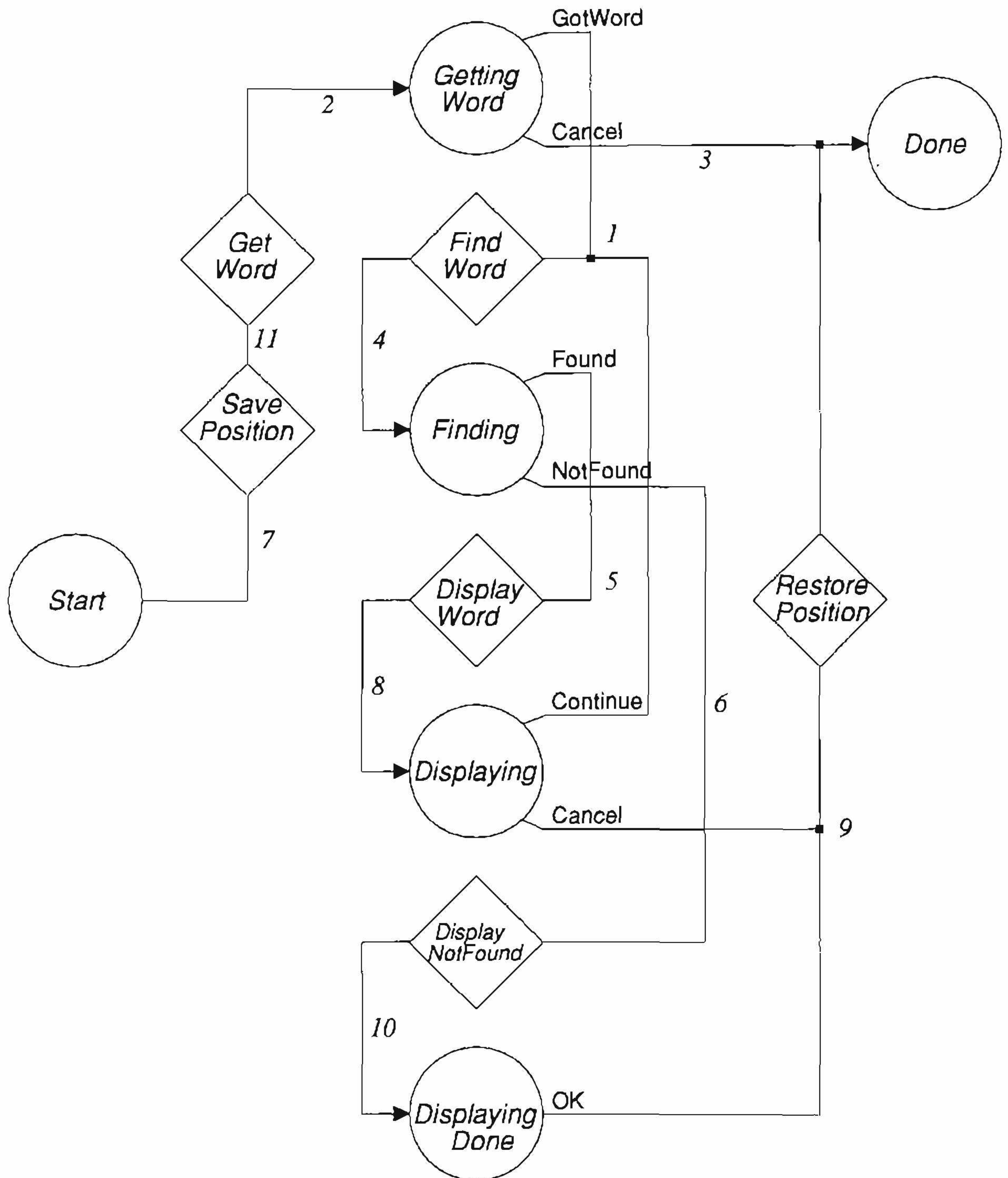


Figure 4.6. Vista Specification as State Transition Diagram

```
BEGIN

     Transition vv9("9");
     Transition vv8("8");
     Transition vv7("7");
     Transition vv6("6");
     Transition vv5("5");
     Transition vv4("4");
     Transition vv3("3");
     Transition vv11("11");
     Transition vv10("10");
     Transition vv2("2");
     Transition vv1("1");

BEGIN_CELLS

    WordFinderGettingWordState("GettingWord", vv2, vv1, vv3);
    WordFinderDoneState("Done", vv3);
    WordFinderAction("GetWord", vv2, vv11);
    WordFinderAction("FindWord", vv4, vv1);
    WordFinderAction("SavePosition", vv11, vv7);
    WordFinderFindingState("Finding", vv4, vv5, vv6);
    WordFinderStartState("Start", vv7);
    WordFinderAction("DisplayWord", vv8, vv5);
    WordFinderAction("RestorePosition", vv3, vv9);
    WordFinderDisplayingState("Displaying", vv8, vv1, vv9);
    WordFinderAction("DisplayNotFound", vv10, vv6);
    WordFinderDisplayingDoneState("DisplayingDone", vv10, vv9);

END_CELLS

END
```

Figure 4.7. Netlist for a Finite State Machine Specification

```
class WordFinderState : public State {
public:
  WordFinderState(const char* name) {
    put_state(this, name); // so this can be found by name
  }
  virtual const char* StateName(void) const {
    return "WordFinderState";
  }
  virtual void GotWord(WordFinder& c) {
    cerr << c << "No transition from GotWord" << endl;
  }
  virtual void Cancel(WordFinder& c) {
    cerr << c << "No transition from Cancel" << endl;
  }
  virtual void Found(WordFinder& c) {
    cerr << c << "No transition from Found" << endl;
  }
  virtual void NotFound(WordFinder& c) {
    cerr << c << "No transition from NotFound" << endl;
  }
  virtual void OK(WordFinder& c) {
    cerr << c << "No transition from OK" << endl;
  }
};
```

Figure 4.8. C++ Code Generated for a *Control* Base Class

```
class WordFinderStateStart : public WordFinderState {
public:
  virtual void Start(FSMContext& c) {
    c.EnterState("GettingWord"); c.SavePosition(); c.GetWord();
  }
} Start("Start");

class WordFinderStateGettingWord : public WordFinderState {
public:
  virtual void GotWord(WordFinder& c) {
    c.EnterState("Finding"); c.FindWord();
  }
  virtual void Cancel(WordFinder& c) {
    c.EnterState("Done");
  }
} GettingWord("GettingWord");

class WordFinderStateDone : public WordFinderState {
public:
} Done("Done");

class WordFinderStateFinding : public WordFinderState {
public:
  virtual void Found(WordFinder& c) {
    c.EnterState("Displaying"); c.DisplayWord();
  }
  virtual void NotFound(WordFinder& c) {
    c.EnterState("DisplayingDone"); c.DisplayNotFound();
  }
} Finding("Finding");

class WordFinderStateDisplaying : public WordFinderState {
public:
  virtual void Continue(WordFinder& c) {
    c.EnterState("Finding"); c.FindWord();
  }
  virtual void Cancel(WordFinder& c) {
    c.EnterState("Done"); c.RestorePosition();
  }
} Displaying("Displaying");

class WordFinderStateDisplayingDone : public WordFinderState {
public:
  virtual void OK(WordFinder& c) {
    c.EnterState("Done"); c.RestorePosition();
  }
} DisplayingDone("DisplayingDone");
```

Figure 4.9. C++ Code Generated for *Control* Derived Classes

```
class WordFinder : public FSMContext {
public:
  WordFinder(void) {
    EnterState("Start");
    Start();
  }
  WordFinderState& GetState(void) {
    return ((WordFinderState&)*itsState);
  }
  void SavePosition(void) {
    // ...
  }
  void GetWord(void) {
    // ...
  }
  void FindWord(void) {
    // ...
  }
  void DisplayWord(void) {
    // ...
  }
  void DisplayNotFound(void) {
    // ...
  }
  void RestorePosition(void) {
    // ...
  }
};
```

Figure 4.10. The *Context* Class for a Word Finder FSM

CHAPTER 5

APPLICATION

In this chapter are presented several specific tool-agent interaction problems to which Vista is applicable, organized by protolib (those itemized in Chapter 3) and exhibited roughly in increasing order of complexity. The wide variety of examples, both real and contrived, is meant to demonstrate the versatility of Vista protolib protocols. As in the last chapter, code and commentary are interwoven with the aim of elucidating the salient features of these vistafications. For the benefit of the curious, the details of the Vista class libraries are found in Appendix A.

To begin, dichotomizing the tools used in the examples in this chapter yields two broad categories, namely:

1. Filters, and
2. Engines.

Filters serve to condense data to manageable proportions, since when information is voluminous, its compression is highly desirable, if not absolutely essential. The Unix **sed** and **grep** programs exemplify filters. Engines do computational work more involved than simple filtering, but, like filters, are usually invoked non-interactively (batch mode) although many allow interactive invocation as well. Spreadsheets and simulators are representative of interactive computation engines.

To refine the first category, a tool must have certain characteristics to be considered a filter, at least one fit to participate in a Unix pipeline. Whimsically adapted from [101], Figure 5.1 presents the prerequisites for Unix filters in directive form.

The ninth commandment is broken by many would-be filters, for example, **ps** (process status), whose author apparently thought it necessary to label each columnar field in a one-line heading. Displaying the state of Unix processes, **ps** varies the number of fields and hence the level of detail depending on the switches, or options supplied to it. Actually, **ps** violates the first commandment as well, since it does not read from standard in. Instead, it queries the Unix kernel to procure its process information. Similarly, **ls** queries the Unix file system, and, via a low-level device driver, reads from a disk to obtain its input. However, since these two tools (among others) write their output to standard out, they can still participate in a pipeline, the only restriction being that they must be at its head, not in the middle nor bringing up the rear[1].

[1]Actually, the Unix shell does not disallow putting **ls** or **ps** at the end of a pipeline. Doing so

I. Thou shalt obtain thy input from standard in.

II. Thou shalt produce thy output on standard out.

III. Thou shalt send any error or diagnostic messages to the standard error output.

IV. Thou shalt not require an input file to be specified on the command line.

V. Thou shalt likewise not require specification of an output file.

VI. Thou shalt make no attempt to interpret thy input data as instructions or commands to thyself.

VII. Thou shalt not ask thy user for additional parameters beyond those supplied on the command line.

VIII. Thou shalt perform a well-defined transformation on thy input.

IX. Thou shalt not festoon thy output with extraneous headers, footers or other formatting.

X. Thou shalt deliver thy output records one per line, and separate their fields with a space, tab, colon or other delimiting character.

Figure 5.1. Ten Commandments for Filters

5.1 Files, Streams and Pipes

Being the main repositories of permanent and temporary data, files figure heavily in Unix. The stream paradigm regulating serial access to this data is another heavyweight, leveraged to maximum advantage in the C++ iostream idiom. Pipes are another mechanism of high utility and renown. Encapsulating these basic Unix services in higher-level protocols is the purpose of the **fsp** protolib.

In yet another dose of déjà vu, Figure 1.1, the simple filtering of a directory listing into a single number giving the count of lines in the listing, is recruited once more. This overworked example will again introduce one of these protocols, that being the non-fictitious counterpart to the one presented in Section 4.2.

The vistafication named `lswc` utilizes a special *pipe* type wire in conjunction with two simple cell types, *UnixO* and *UnixI*, thus:

```
// File: lswc.c

#include "lswc.h" // design
#include "fsp.h" // protolib
#include "UnixO.h" // (NAME, OUTPUT(pipe, vvout))
outclude
#include "UnixI.h" // (NAME, BIDIR(pipe, vvin))
outclude
```

---

results in no error exit, merely the loss of all upstream output.

```
BEGIN

     pipe vv1("1");

BEGIN_CELLS

    UnixO("ls -l", vv1);
    UnixI("wc -l", vv1);

END_CELLS

END
```

Unlike in the visually and topologically similar **sig** test example of Section 4.3, there is no need for multiple cell instantiations in this simple pipeline, hence `BEGIN_CELLS` and `END_CELLS` do not implement a *do while fsp IsVistafying* loop, but instead expand to nothing:

```
// File: lswc.h

#define BEGIN MAIN_BEGIN

#define BEGIN_CELLS

#define END_CELLS

#define END MAIN_END
```

The single-ported *UnixI* and *UnixO* cell classes, shown below, are simplicity incarnate, doing nothing other than supplying their names (Unix commands) to the >> and << operators of *pipe*, whose class definition appears in its entirety in Figure 5.2:

```
defCell(UnixI, (NAME, INPUT(pipe, in)))
  in >> name;

defCell(UnixO, (NAME, OUTPUT(pipe, out)))
  out << name;
```

The *fsp* Vistafier has no work to do at all, as it is all done by the symmetrical *pipe* wire, which encapsulates two streams, one for input and one for output, as shown. The *istream* and the *ostream* members each use a special-purpose buffer called a *procbuf* (process buffer), which is a subclass of *filebuf* which in turn inherits from the base *streambuf* class in the C++ iostream library (the GNU version from the FSF). The purpose of *procbuf* is to provide similar functionality to the Unix

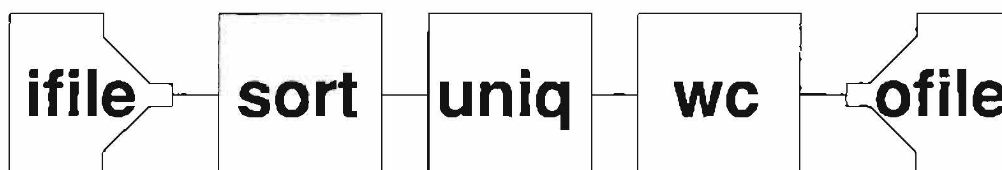
```
class pipe : public Wire {
private:
  procbuf ipb;
  istream* is;
  procbuf opb;
  ostream* os;
  unsigned char c;
public:
  pipe(const char* name) : Wire("pipe", name), is(nil), os(nil) {
  }
  boolean Get(void){
    return (is != nil && is->get(c) && !is->eof());
  }
  void Got(void) {
    *os << c;
  }
  void Flow(void) {
    while (Get())
      Got();
  }
  void operator << (const char* command) {
    if (ipb.open(command, ios::in)) {
      is = new istream(&ipb);
      if (os != nil) {
        Flow();
      }
    }
  }
  void operator >> (const char* command) {
    if (opb.open(command, ios::out)) {
      os = new ostream(&opb);
      if (is != nil) {
        Flow();
      }
    }
  }
  ~pipe(void) {
    delete is;
    delete os;
  }
};
```

Figure 5.2. The *pipe* Class Definition

system calls **popen** and **pclose**, which open or close a (Unix) pipe for I/O from or to a process.[2]

Note that both << and >> operators check for a null pointer to the opposite stream, only calling *Flow* if it is non-null. This allows either cell to be instantiated first, the flow from the *UnixO* command to the *UnixI* command starting only after the second cell is instantiated. The *Get/Got* flow is an inefficient character-at-a-time read/write, but could easily have been done line-at-a-time.

These two cell classes are not terribly useful, as they only allow for either input or output, not both. Thus, a *UnixO* cell can only be at the head of a two-command-only pipeline, while the *UnixI* cell can only be at its tail. The double-ported *UnixIO* cell class, in conjunction with the *StdIO* wire class, overcomes these limitations at the cost of more complexity. Representative usage of these classes is shown in the following pipeline, whose ends are fitted with two additional cells of type *UnixIFile* and *UnixOFile*:



This file-capped pipeline, which simply puts into a specified output file a count of the unique (no duplicate lines) contents of a specified input file, translates into the following shell syntax:

```
sort < $ifile | uniq | wc > $ofile
```

In this shell expression, the `ifile` and `ofile` shell variables store the names of the files. The cell wrappers for *UnixIFile* and *UnixOFile* display these variable names, but the cell *instance names* are the file name values of these two variables. These two wrappers have a non-rectangular (and mirrored) shape to distinguish them from (each other and) the *UnixIO* cells, whose instance names are the actual Unix commands to be invoked, including any arguments, although only the command name itself is displayed in their wrappers.

It is interesting to note that the use of these two special cells to specify input and output file redirection is not strictly necessary. At the cost of an extra Unix process apiece, the same pipeline can be rendered in five *UnixIO* cells, whose corresponding shell rendition, given input and output files named *foo* and *oof* respectively, is:

```
cat foo | sort | uniq | wc | tac oof
```

The **tac** command is a one-line shell script counterpart to the **cat** command, and in fact does `cat > $1` which merely invokes **cat** and does the ">" Unix output

---

[2] A call to **popen** returns an open **FILE*** stream pointer (which **pclose** closes), the writing to which or reading from which is the same as writing to the standard input, or reading from the standard output of a Unix command. Calling the *procbuf::open* method also creates a pipe between the calling process and the named command, but allows C++ iostream-based standard I/O writing and reading.

redirection to its argument-named file. The standard input of **tac** is connected to the standard input of **cat** by the shell.

Hiding the redirection and using one extra process (actually two, since the **tac** script runs in a shell process too), plus the additional **cat** at the head of this pipeline, is an expensive way to achieve uniformity. Uniformity and parsimony aside, even using the more efficient I/O file cells provides no real advantage over directly using the Unix shell, which does the same kind of pipe creation, file redirection and process invocation that Vista does, only better.[3] Therefore, since the shell can easily interpret these kinds of pipelines without the substantial overhead of an intervening visual framework, the question begs, of what value is Vista here? As noted in Section 2.1 and shown there in Figure 2.3 and here in Figure 5.3, the value added by Vista comes in the form of support for *multifurcated* pipelines, whose data streams can be forked and joined in various useful ways.

Compared to *UnixIO*, the significantly higher complexity of doing *multiple* I/O with pipes merits its encapsulation in the *UnixMIO* class, which makes use of all three Analyzing, Manifesting and Executing lifetimes to do its job, which is explained briefly shortly.

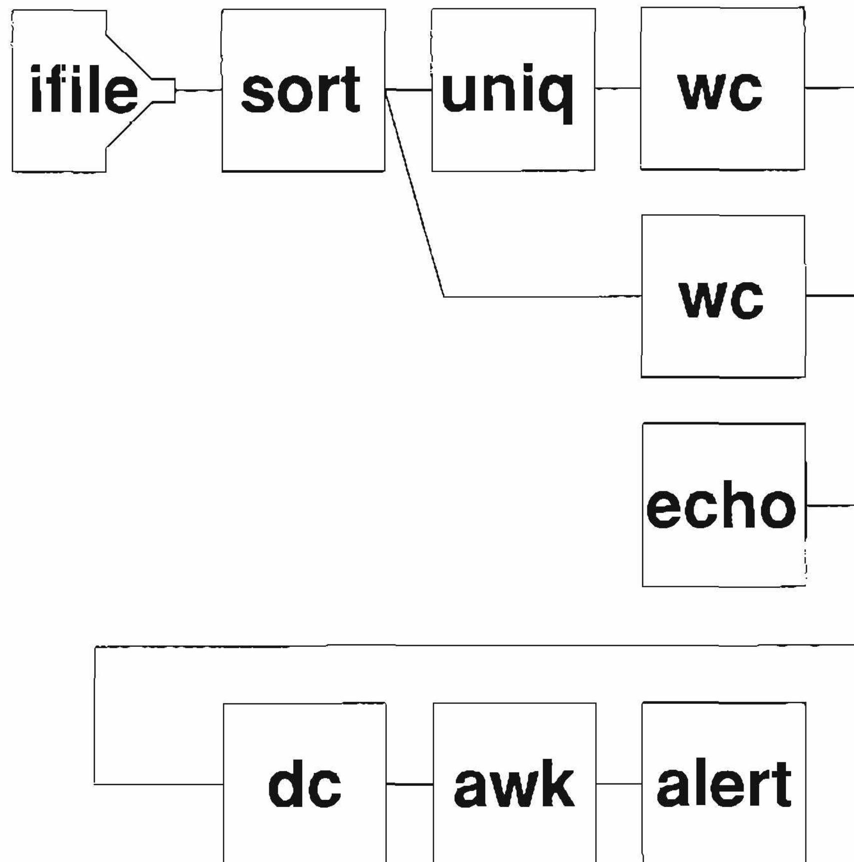
In the vistafication of Figure 5.3, the (square) cells are all instances of the *UnixMIO* class. Building on the above simple pipeline, (but avoiding the need that the shell would have for temporary, intermediate files), what this code does is tell the user how many duplicate lines appear in the given input file. Forking the output from **sort** into another **wc**, that count and the **uniq** count are joined with a simple instruction supplied by **echo** to become the input of **dc** (desk calculator), which computes the difference of the two counts. The output of **dc** is then fed to **awk**, which formats it and pipes it to **alert**, which in turn pops up a dialog box containing the message read from its standard input, plus a pushbutton with which to dismiss the **alert** window.

Note in this (decapitated and folded) vistafication that the wire instance `::vvin` is invisibly connected to the input port of the echo cell. Likewise is `::vvout` connected to the output of alert. As mentioned in Subsection 3.4.2 on page 33, unwired ports are codified with the port name prefixed by an identifying marker, which by design is the `::` C++ scope resolution operator. It is perfectly legal to leave unconnected the standard input or standard output of these (and many other) Unix commands. Thus, this will not be flagged as a connectivity error by the Analyzer, but only because the vistafication *is* legal C++ code, as the **fsp** library predefines `vvin` and `vvout` as global instances of the *StdIO* wire class, and knows how to deal with them differently than with the explicit locally-allocated instances.

While Analyzing, the *UnixMIO* cells simply increment the fanin and fanout counts of their attached wires, so that, while Manifesting, the appropriate number of Unix pipes can be created. The file descriptors of the reading and writing ends of these pipes are stored in separate arrays in the *StdIO* wire instances, as are the names of any input or output files. It is in the final Executing stage where the Unix fork/exec idiom is called upon to create the cell-named processes, attaching

---

[3]The **fsp** class libraries implement essentially what the shell does, only using a compiled C++ program rather than parsing and interpreting a textual expression.




```
BEGIN
     StdIO vv6("6"); StdIO vv2("2"); StdIO vv3("3");
     StdIO vv1("1"); StdIO vv4("4"); StdIO vv5("5");
BEGIN_CELLS

    UnixIFile("howmanydups.in", vv1);
    UnixMIO("sort", vv1, vv2);
    UnixMIO("uniq", vv2, vv3);
    UnixMIO("wc -l", vv3, vv4);
    UnixMIO("wc -l", vv2, vv4);
    UnixMIO("echo -pq", ::vvin, vv4);
    UnixMIO("dc", vv4, vv5);
    UnixMIO("awk -f howmanydups.awk", vv5, vv6);
    UnixMIO("alert", vv6, ::vvout);

END_CELLS
END
```

Figure 5.3. The howmanydups Vistafication

their standard input and output to the corresponding wires' writing or reading pipe ends, or redirecting them to a file, and closing any unneeded pipe ends. For informational (and debugging) purposes, the id and command name/arguments of each process are printed out when it is created. Thus, sample execution of the above vistafication reveals that it really creates ten rather than eight processes:

```
10645 dispatch fork 23 -8 -14
10646 sort
10647 uniq
10648 wc -l
10649 wc -l
10650 echo -pq
10651 dispatch join 11 15 17 -7
10652 dc
10653 awk -f howmanydups.awk
10654 alert
```

Considered a crucial component of the **fsp** library, the **dispatch** command is the agent that handles the actual forking or joining of pipes between processes.[4] The numeric arguments to **dispatch** represent the input and output file descriptors, which are distinguished by the latter being negative.[5] The first **dispatch** arranges for a *broadcast redistribution* to file descriptors 8 and 14 of all data coming from file descriptor 23, which identifies the reading end of the pipe being written to by the **sort** process. The second **dispatch** writes to file descriptor 7 (input to **dc**) in round-robin fashion the data coming down the pipes identified by descriptors 11, 15 and 17. Figure 5.4 shows how the above multifurcated pipeline would appear were it necessary for the user to insert these special dispatch cells manually. Fortunately for the user, it is not.

The **pipe** Unix system call creates the actual interprocess communication channel, in which are buffered up to a fixed number (e.g., 4096) of bytes inserted by the writing process before the writing process is blocked, waiting for the reading process to extract the buffered bytes, which it must do in FIFO order. Whenever such low-level mechanisms can be used in this way to connect arbitrary processes, there is a danger of deadlock. In fact, in the "BUGS" section of the Unix "man" page for the **pipe** system call, it states: *Should more than 4096 bytes be necessary in any pipe among a loop of processes, deadlock will occur.* This is in fact a sufficient but unnecessary condition for deadlock. Figure 5.5 shows a single-process loop configuration that just as surely will deadlock with zero bytes in the pipe buffer.

---

[4]Transparently interposed between cells by the *UnixMIO* protocol, the **dispatch** program relies on the Unix **select** system call for doing this *synchronous I/O multiplexing*. The InterViews 3.1 *Dispatch* library elegantly encapsulates **select** in a convenient interface that uses a *Dispatcher* class object that notifies *IOHandler* class objects when input or output is ready on their associated file descriptors.

[5]On the command line they are negative numbers, but their absolute values are used for the actual values of the file descriptors needed by **dispatch**.

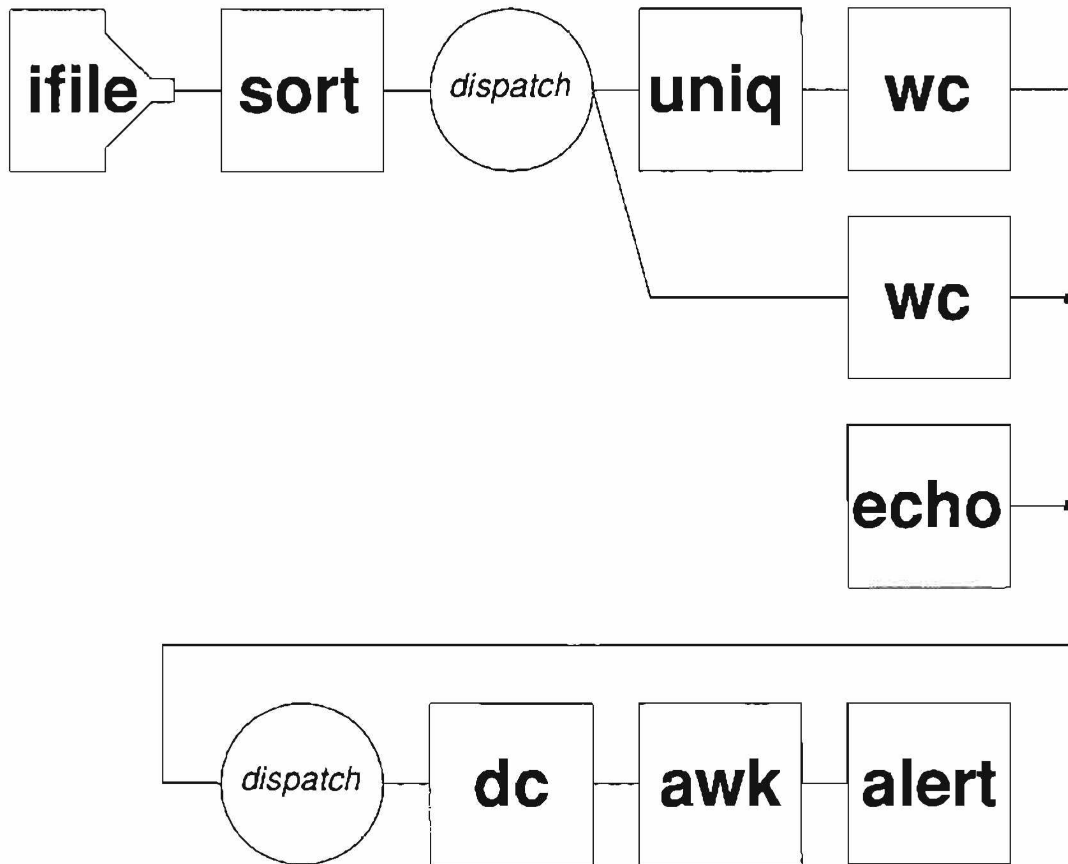


Figure 5.4. Explicit Dispatch Cells

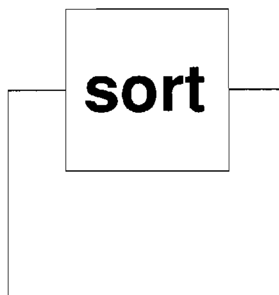


Figure 5.5. Sort Deadlock

Moreover, other cyclically-connected groups of cells will also deadlock if they all need to read their standard input before writing anything to their standard output. For a two-cell example, **sort** feeding **uniq** feeding back to **sort**, as in (just the bare text this time):

```
BEGIN

     StdIO vv2("2");
     StdIO vv1("1");

BEGIN_CELLS

    UnixMIO("sort", vv1, vv2);
    UnixMIO("uniq", vv2, vv1);

END_CELLS

END
```

The rules for the **fsp** protolib should disallow such self-connected configurations, and so they do. Here are three key rules that incorporate into the validation query:

```
connected(Ca,Cb) :- oconnected(Ca,W), iconnected(Cb,W).
connected(Ca,Cb) :- connected(Ca,Cx), connected(Cx,Cb).
selfconnected(C) :- connected(C,C).
```

The first rule says that cell `Ca` is connected to cell `Cb` if wire `W` is connected to the output of `Ca` and the input of `Cb`. The second rule is recursive, expressed in terms of itself, stating that two cells are considered connected if each is connected to a common third cell. That is, connectivity is a transitive relation. The third rule succinctly describes a cell connected to itself.

Per-process open file descriptors being a limited Unix kernel resource, there must also be rules to constrain the fanin or fanout of *StdIO* wires, although the upper limit is high enough (e.g., 64) that most practical applications never even approach it. While it matters whether it is an input or an output port a wire is attached to, mainly it is wire *terminals* that matter to the fanin/fanout count determination, since fanin and fanout are properties of wires, not ports. A wire that forks has a

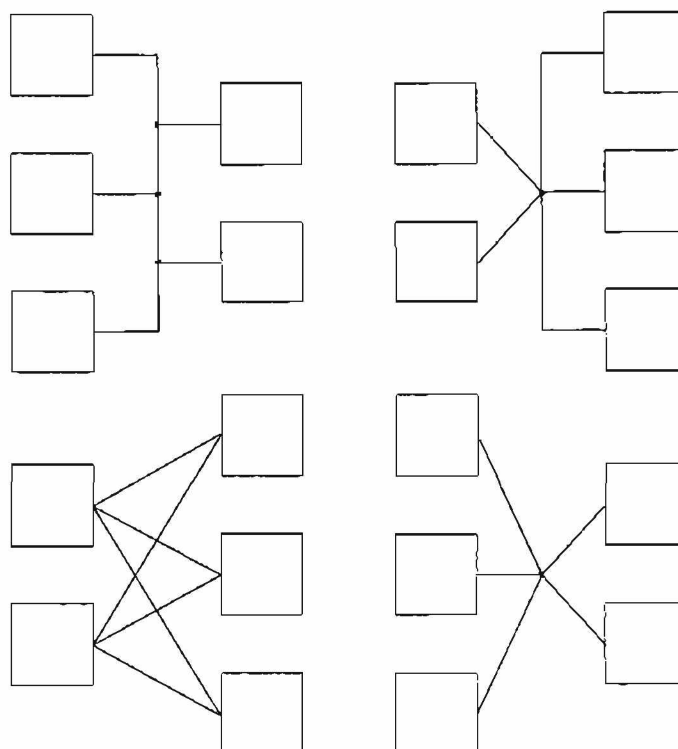


Figure 5.6. Wires That Fork *and* Join

fanout greater than one, while a wire that joins has a fanin greater than one. The *UnixMIO* cell protocol can deal with wires that fork but do not join, wires that join but do not fork, or wires that neither fork nor join. The fourth case (see Figure 5.6 for four visual depictions) of a wire that both forks and joins is not handled, and therefore must be proscribed. It is quite difficult to come up with pipelines where this makes sense, however, so this is not a severe restriction.

More examples of pipeline multifurcation will be forthcoming. Before moving on to the next Vista protolib, another interesting protocol instrumented by **fsp** is exemplified by the design shown in Figure 5.7, and again slightly modified in Figure 5.10. These two examples demonstrate a variety of versatile features:

- dataflow using both bundled (hierarchical) and unbundled wires,
- cell hierarchy and structural modeling,
- cell metamorphosis from structural to behavioral,
- two-way pipe/stream communication and synchronization,
- compute/output loop implementation,
- command and data files, and
- interaction with **wish** (see Subsection 2.5.1).

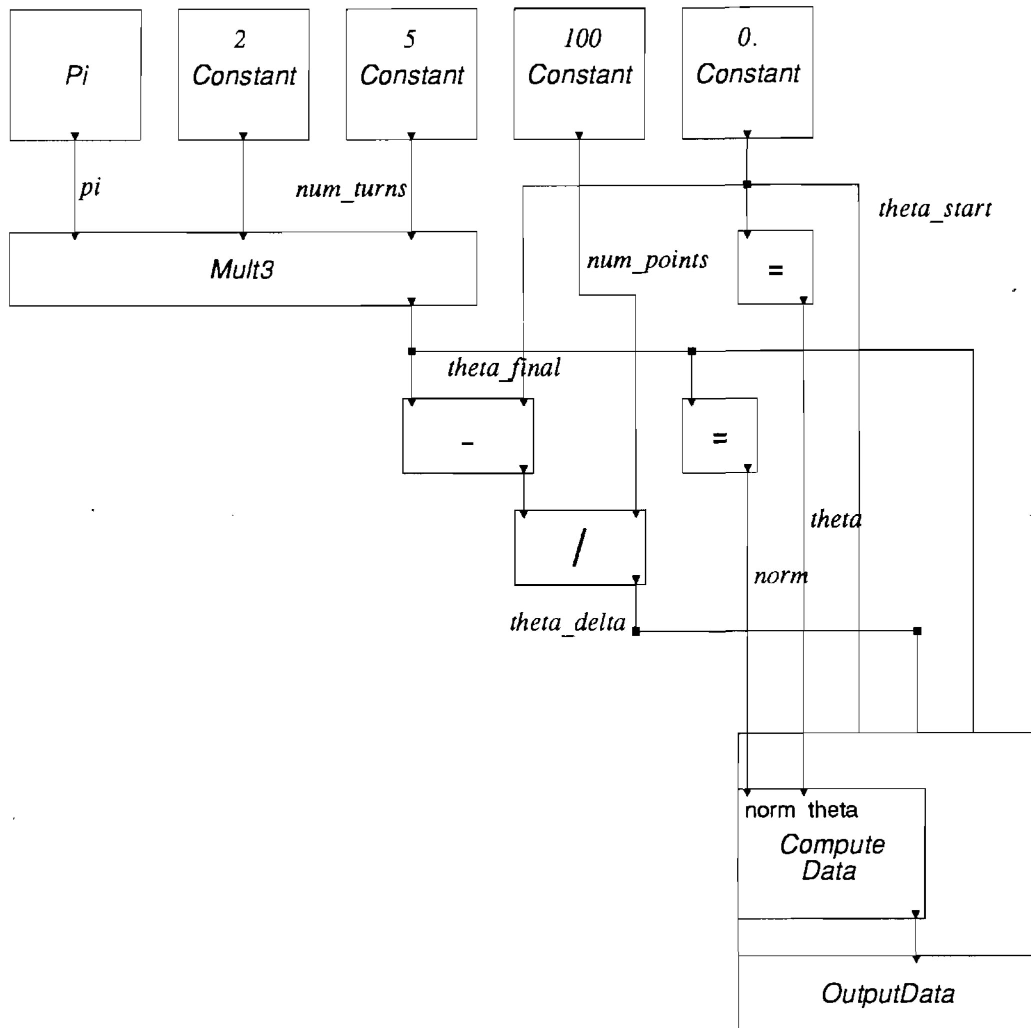


Figure 5.7. The `gnuplot` Schematic

The vistafied agent interacts with the **gnuplot** tool, which is a command-driven interactive function plotting program. Their interaction is coordinated by an object (named `uipmaster`) of type *UnixInteractiveProgramMaster*, which acts as a kind of surrogate Vistafier for this vistafication, whose preamble appears in Figure 5.8 and whose body is shown in Figure 5.9.

In this example, a special setup is required in the `gnuplot.h` file in order to initialize the `uipmaster` object and have it invoke **gnuplot**:

```
// File: gnuplot.c

#include "gnuplot.h" // design
#include "fsp.h" // protolib
#include "Pi.h" // (NAME, OUTPUT(abc, vvout))
outclude
#include "Constant.h" // (NAME, OUTPUT(abc, vvout))
outclude
#include "Mult3.h" // (NAME, INPUT(abc, vvi3), INPUT(abc, vvi2),
                             INPUT(abc, vvi1), OUTPUT(abc, vvout))
outclude
#include "Assign.h" // (NAME, INPUT(abc, vvin), OUTPUT(abc, vvout))
outclude
#include "Sub2.h" // (NAME, INPUT(abc, vvin2), OUTPUT(abc, vvout),
                            INPUT(abc, vvin1))
outclude
#include "Div2.h" // (NAME, INPUT(abc, vvin2), OUTPUT(abc, vvout),
                            INPUT(abc, vvin1))
outclude
#include "ComputeData.h" // (NAME, INPUT(abc, vvnorm),
                                   INPUT(abc, vvtheta),
                                   OUTPUT(xyz, vvout))
outclude
#include "OutputData.h" // (NAME, INPUT(xyz, vvin))
outclude
#include "DataLoop.h" // (NAME, INPUT(abc, vvi),
                                INPUT(abc, vvd),
                                INPUT(abc, vvf),
                                INPUT(abc, vvtheta),
                                INPUT(abc, vvnorm))
BEGIN_DataLoop
       xyz vv1("1");
BEGIN_DataLoop_CELLS
      ComputeData("compute", vvnorm, vvtheta, vv1);
      OutputData("putout", vv1);
END_DataLoop_CELLS
END_DataLoop
outclude
```

Figure 5.8. The `gnuplot` Vistafication Preamble

```
BEGIN

     abc vvpi("pi");
     abc vvtheta("theta");
     abc vvnum_points("num_points");
     abc vv1("1");
     abc vvnum_turns("num_turns");
     abc vvtheta_final("theta_final");
     abc vv2("2");
     abc vvnorm("norm");
     abc vvtheta_delta("theta_delta");
     abc vvtheta_start("theta_start");

BEGIN_CELLS

    Pi("pi", vvpi);
    Constant("2", vv1);
    Constant("5", vvnum_turns);
    Constant("100", vvnum_points);
    Constant("0.", vvtheta_start);
    Mult3("a", vvnum_turns, vv1, vvpi, vvtheta_final);
    Assign("b", vvtheta_start, vvtheta);
    Sub2("c", vvtheta_start, vv2, vvtheta_final);
    Assign("d", vvtheta_final, vvnorm);
    Div2("e", vvnum_points, vvtheta_delta, vv2);
    DataLoop("f", vvtheta_start, vvtheta_delta,
                  vvtheta_final, vvtheta, vvnorm);
END_CELLS

END
```

Figure 5.9. The gnuplot Vistafication Body

```
// File: gnuplot.h

#define BEGIN \
int main(int argc, char** argv) {\
  int synch_time = 0;\
  if (argc >= 6) synch_time = atoi(argv[5]);\
  uipmaster.Init("/usr/local/gnu/bin", "gnuplot",\
                 "pause 0", "paused",\
                 "quit", synch_time);

          #define BEGIN_CELLS \
            uipmaster.SendCommand("load \"gnuplot.init\"");

          #define END_CELLS \
            uipmaster.Pause("alert", "Done");

          #define END \
            return uipmaster.Exit(); }
```

The *Init* method needs four arguments besides the name of the interactive program and the directory where its executable binary file resides (arguments two and one, respectively). These four are:

1. a *pause* command,
2. an *expected reply* therefrom,
3. a *quit* command, and
4. a *synchronization time* argument.

The pause command is combined with the expected reply, at least for the **gnuplot** program, and it is this reply that is written by the program to its standard output (or standard error output) after it pauses for a specified length of time when given the pause command. Thus, telling **gnuplot** to "pause 0 paused" makes it return the message "paused" after a wait of zero seconds, that is, immediately.

The gist of the two-way communication and synchronization is as follows. Before forking off the child **gnuplot** process, the *Init* method (running in the parent process) creates two pipes, one for parent-to-child (*ptoc*) communication, and the other for child-to-parent (*ctop*) communication. The output of the *ptoc* pipe will act as the standard input for **gnuplot**, and the input of the *ctop* pipe will act as its standard output. The *SendCommand* method calls *PutCommand*, which writes a **gnuplot** command to the *ptoc* pipe, and then calls the *Synch* method. If the specified *synch_time* variable is zero, *Synch* returns immediately, meaning no synchronization is attempted. Otherwise, it calls *PutCommand* with the pause command, and then enters a loop that exits only when the expected reply is read from the *ctop* pipe that will be written by **gnuplot**, sleeping for *synch_time* microseconds between unsuccessful read attempts.

The *UnixInteractiveProgramMaster::Pause* method is unrelated to the above two-way tool-agent handshake. In the call shown above, it uses the **alert** program to send a message to the *user* instead of **gnuplot**. It then waits for the user to dismiss the dialog box before returning. Thus, this agent-user interaction provides an opportunity for the user to pause the tool-agent interaction indefinitely, in order to examine the function plotted by **gnuplot** before it goes away.

The *Exit* method simply returns zero after calling *PutCommand* with the quit command, which tells **gnuplot** to close its window and exit.

Before any cells are instantiated, the `uipmaster` sends the "load" command to **gnuplot** telling it to read and execute the following setup commands from the `gnuplot.init` file:[6]

```
set terminal X11
set xrange [-1:1]
set yrange [-1:1]
set zrange [0:1]
set parametric
clear
```

The "clear" command erases the current screen or output device as specified by the "set terminal" command. In this case, clearing the X11 window "terminal" causes it to be mapped and exposed, and ready to accept drawing requests. The "'set range" commands explicitly set the ranges that will be displayed for the three axes, turning off their default automatic scaling to fit the data being plotted. The "set parametric" command changes the meaning of "plot" (and "splot") from normal (single-valued) functions to parametric functions.

The graph being plotted is a 3-dimensional spiral, hence a triplet of parametric functions is required. The parameter controlling these three functions (one each for x, y and z) is *theta*, an angle expressed in radians. The dataflow cells (all but the bottommost) compute the initial values for wires controlling the *theta* parameter, while the bottommost cell handles the loop that increments *theta* and computes and outputs the data points that depend on *theta*.

The dataflow cells operate on wires of type *abc*, which overloads the = and += base *Wire* class operators to take *double* type instead of *int* type arguments. These cells all do what their class name suggests, the one-line contents of most of them being as follows:

Assign:	`vvout = *vvin;`
Div2:	`vvout = *vvin1 / *vvin2;`
Sub2:	`vvout = *vvin1 - *vvin2;`
Mult3:	`vvout = *vvi1 * *vvi2 * *vvi3;`

[6]These six commands could just have easily been sent by separate calls to *SendCommand* in the `BEGIN_CELLS` macro. Using a file as a bundler of commands is convenient, but has a downside as well—the user has one more file to keep track of.

Being hierarchical, the *DataLoop* cell bears closer scrutiny. Shown *clear-wrapped* in Figure 5.7 and *opaque-wrapped* in Figure 5.10, it is shown below with its structural netlist appended. This structural netlist, as generated by the Codifier, appears in Figure 5.8 between the line `#include "DataLoop.h"` and the following `outclude` line.

```
// File: DataLoop.h

#include "abc.h"

#define BEGIN_DataLoop
#define BEGIN_DataLoop_CELLS \
  for (vvtheta = *vvi; *vvtheta < *vvf; vvtheta += *vvd) {
#define END_DataLoop_CELLS \
  }
#define END_DataLoop


defCell(DataLoop, (NAME,
                   INPUT(abc, vvi),
                   INPUT(abc, vvd),
                   INPUT(abc, vvf),
                   BIDIR(abc, vvtheta),
                   INPUT(abc, vvnorm)))
BEGIN_DataLoop
       xyz vv1("1");
BEGIN_DataLoop_CELLS
      ComputeData("compute", vvnorm, vvtheta, vv1);
      OutputData("putout", vv1);
END_DataLoop_CELLS
END_DataLoop
```

The Codifier brackets the wires and cells contained in a hierarchical cell in the same way it brackets the top-level wires and cells, except that it replaces `BEGIN` and `END` with `BEGIN_`*CellClassName* and `END_`*CellClassName* in both sets of macros. Here is the *DataLoop* cell definition with the bracketing macros expanded, showing clearly how the C++ *for* statement is used to implement a loop:

```
defCell(DataLoop, (NAME,
                   INPUT(abc, vvi),
                   INPUT(abc, vvd),
                   INPUT(abc, vvf),
                   BIDIR(abc, vvtheta),
                   INPUT(abc, vvnorm)))
  xyz vv1("1");
```

```
for (vvtheta = *vvi; *vvtheta < *vvf; vvtheta += *vvd) {
   ComputeData("compute", vvnorm, vvtheta, vv1);
   OutputData("putout", vv1);
}
```

Note that the local wire connecting the *ComputeData* and *OutputData* cells is of type *xyz*, which simply serves to bundle together three *double* variables, and shows that wires can be hierarchical too.[7] Here are the definitions of the *xyz* wire and the two cells it connects:

```
// File: xyz.h

class xyz : public Wire {
public:
  double x, y, z;
  xyz(const char* name) : Wire("xyz", name) {
  }
};

// File: ComputeData.h

defCell(ComputeData, (NAME,
                      INPUT(abc, vvnorm), INPUT(abc, vvtheta),
                      OUTPUT(xyz, vvout)))
  vvout.z = *vvtheta / *vvnorm;
  vvout.x = vvout.z * cos(*vvtheta);
  vvout.y = vvout.z * sin(*vvtheta);

// File: OutputData.h

defCell(OutputData, (NAME, INPUT(xyz, vvin)))
  uipmaster.GetOutputFileStream()
     << vvin.x << ' ' << vvin.y << ' ' << vvin.z << endl;
  uipmaster.OutputLineCount()++; // increment count of output lines
  if (uipmaster.OutputLineCount() == 2) { char msg[100];
    sprintf(msg, "splot \"%s\" w lines",
            uipmaster.GetOutputFileName());
    uipmaster.SendCommand(msg);
  } else if (uipmaster.OutputLineCount() > 2) {
    uipmaster.SendCommand("replot");
  }
```

[7]The flow between the compute and output cells could also have been achieved with three unbundled (non-hierarchical) wires, but in this example it makes sense to encapsulate the trio, as they represent a single spatial data point.

The "plot" and "splot" commands are the primary commands of the **gnuplot** program. The former plots 2-d functions (mathematical expressions, e.g. $\sin(x)$) and data (read from a file), while the latter plots 3-d surfaces and data. In the form used above, a file name and a drawing style[8] are supplied as "splot" arguments. Both the file name and its associated output stream are maintained by the `uipmaster` by delegation to a member object of type *TemporaryOutputFile*, which in turn encapsulates the operations of creating, opening for writing, closing and deleting a file. This class also frees its clients from the necessity of choosing a name, calling upon the Unix *tmpnam* function to generate a name that can safely be used for a temporary file.[9]

Since the `uipmaster` keeps track of how many lines are written to the output file, when three or more lines have been written it suffices to send **gnuplot** the "replot" command, which repeats the last "plot" or "splot" command. The "splot" command is issued when two $\langle x, y, z \rangle$ triples have been written to the file, which is the minimum number of data points required to draw a line. On each subsequent instantiation of *OutputData* the data in the growing file is replotted, and because the **gnuplot** window is not cleared between replots, the visual effect is an animation of the growing spiral.

The smoothness of the spiral is determined by how many points are plotted, as straight line segments are drawn between successive points, and the total length of the spiral is fixed. How fast it rises in the z direction is determined by the number of (360°) turns it makes as it rises. These two parameters, called *num_points* and *num_turns*, have the constant values of 100 and 5 respectively. Figures 5.11 and 5.12 show the **gnuplot** window for two different sets of values of *num_turns* and *num_points*.

The *Constant* cell class treats the instance name as the value, which is type-converted and assigned to its output wire. In Figure 5.10 the middle two *Constant* cells have been replaced by *Variable* cells, which allow the user to adjust the values of these two parameters at runtime, rather than at compile time. Here is the definition of *Variable*:

```
// File: Variable.h

#include "abc.h"

defCell(Variable, (NAME, OUTPUT(abc, vvout)))
  vvout = NUM uipmaster.GetCommand(name);
```

In this case, the cell name is a Unix command that is required on exit to write a single numeric value to its standard output. The *GetCommand* method opens

[8]Drawing "w lines" means *with lines*, as opposed to *points*, for example.

[9]Safe in the sense that, although a common "scratch" directory (e.g., /usr/tmp) is used, another user calling the same function will get a unique file name that is guaranteed not to clash.

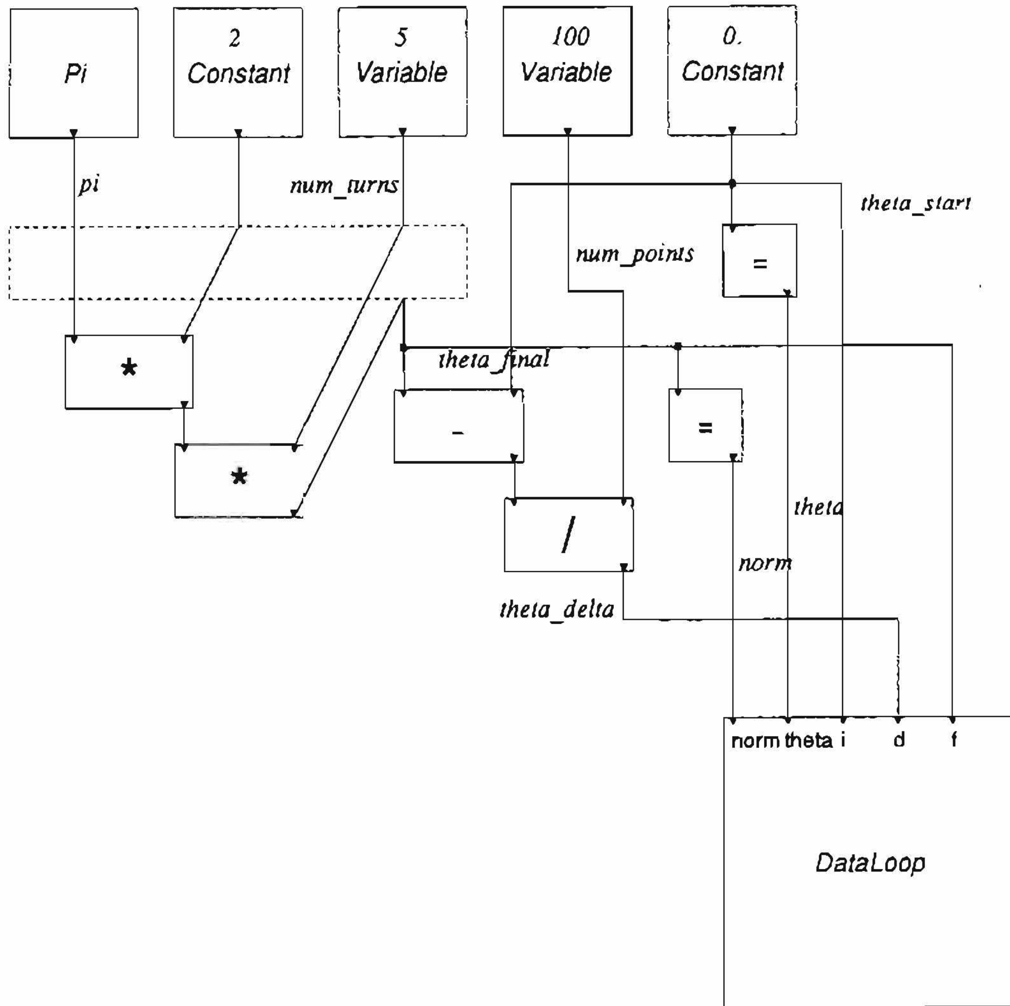


Figure 5.10. The gnuplotvar Schematic

a pipe to the command and reads the pipe to garner and return this value. The *Variable* cells in the gnuplotvar vistafication are:

```
Variable("hscale num_turns  250  1  10  1   5", vvnum_turns);
Variable("hscale num_points 250 50 200 50 100", vvnum_points);
```

The **hscale** program is a **wish** script that creates an X window displaying a horizontal scale, allowing the user to choose a value within the defined range by clicking on the scale, or sliding the indicator along it. Figure 5.13 contains the **hscale** script as well as an example of its use, showing the scale displayed for the second of the above two *Variable* cell calls, and revealing some of the flavor and power of the Tcl/Tk language.

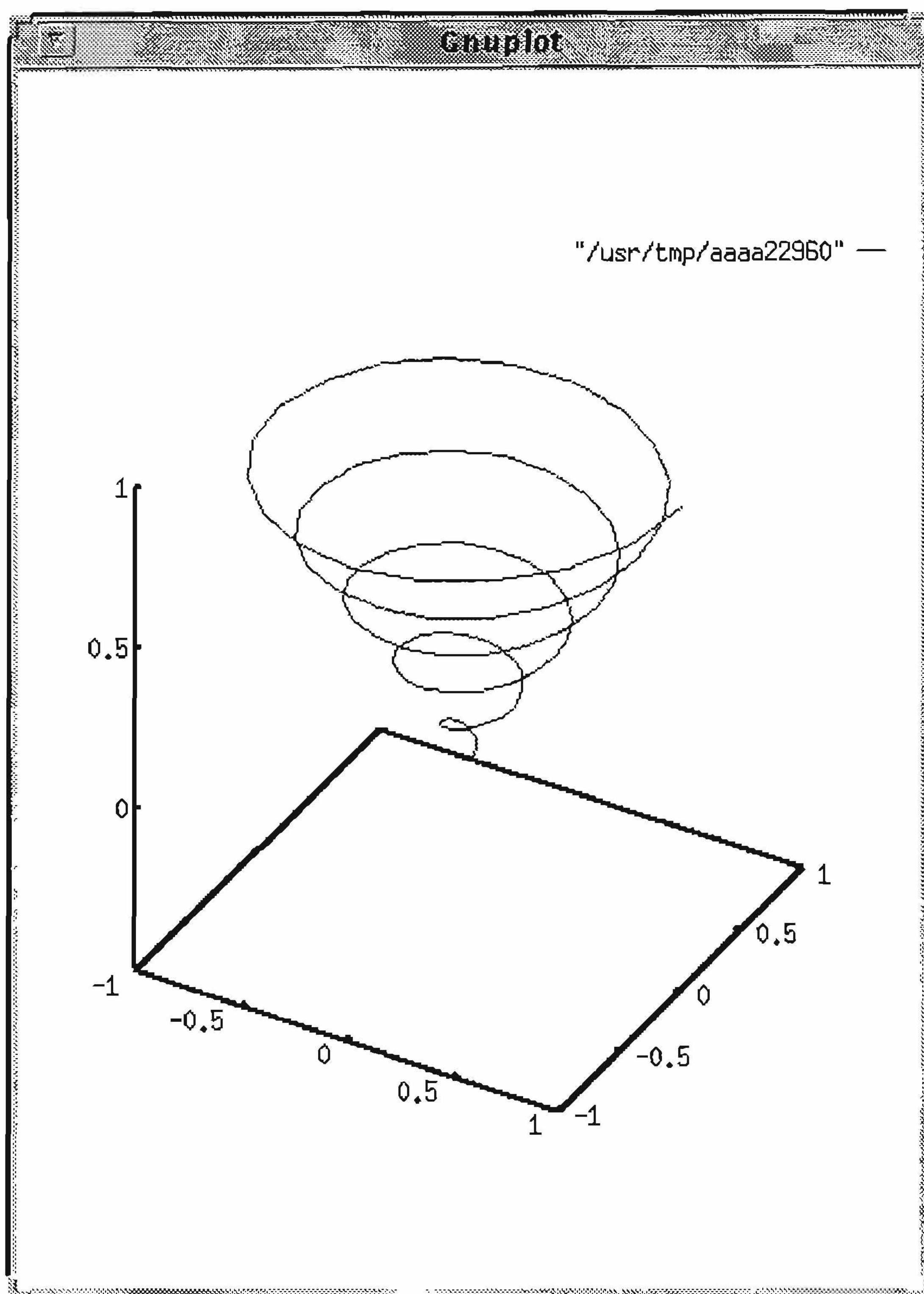


Figure 5.11. A gnuplotted Spiral with 5 Turns and 100 Points

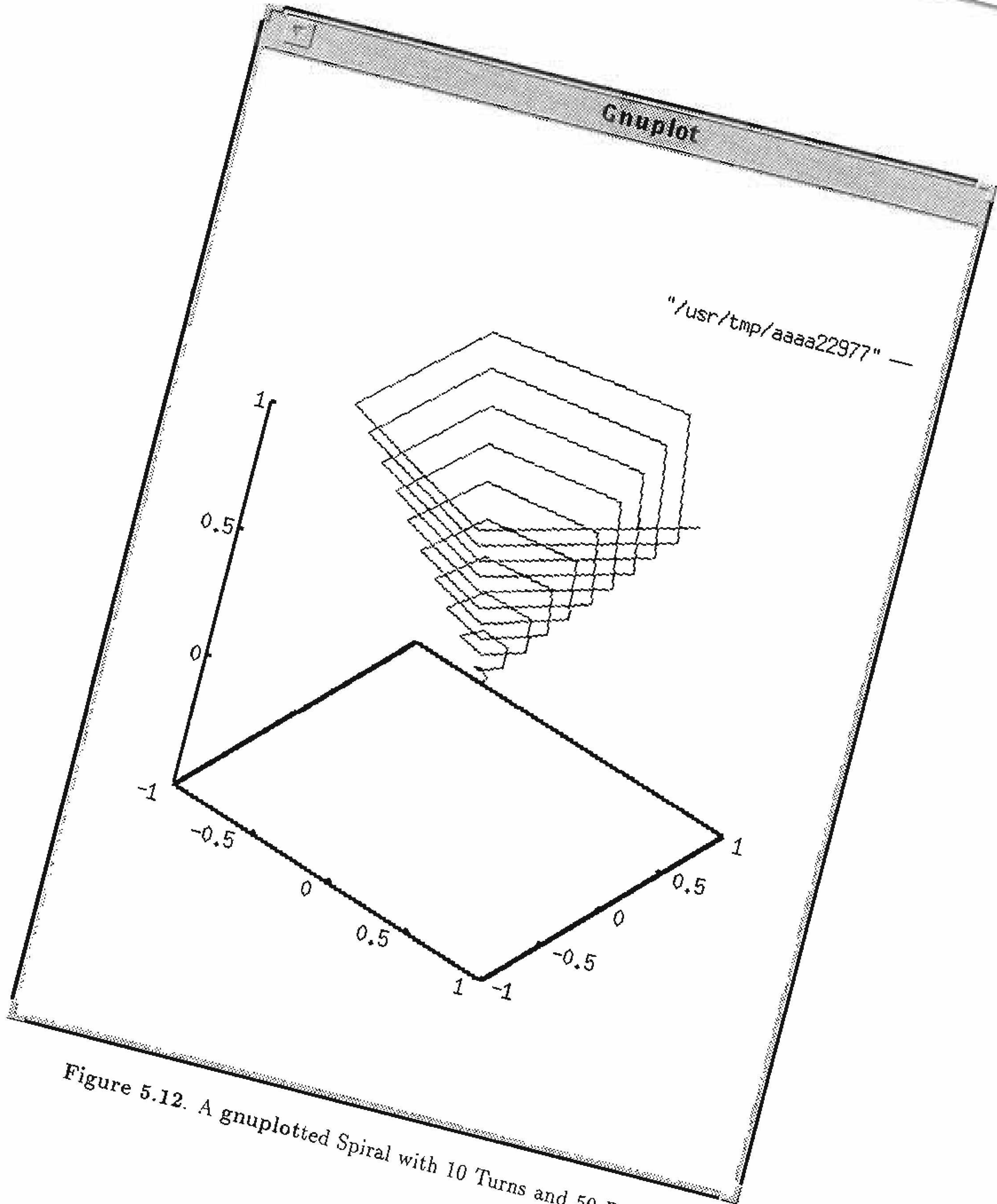


Figure 5.12. A gnuplotted Spiral with 10 Turns and 50 Points

```
proc hscale {variable length from to tickinterval initialvalue} {
  set title "Choose value for "
  append title $variable
  wm title . $title
  wm geometry . +300+300
  scale .scale -orient horizontal -length $length -from $from \
             -to $to -tickinterval $tickinterval
  .scale set $initialvalue
  button .ok -text OK -command "putval .scale"
  pack append . .scale {top fill} .ok {bottom fill}
}
proc putval {s} {
  puts stdout [$s get]
  destroy .
}
if {$argc != 6} {
  puts stdout \
  "Usage: hscale variable length from to tickinterval initialvalue";
  exit 1
}
set v [lindex $argv 0]; set l [lindex $argv 1]
set f [lindex $argv 2]; set t [lindex $argv 3]
set i [lindex $argv 4]; set a [lindex $argv 5]

if {$f >= $t} {
  puts stdout $f
  destroy .
} else {
  hscale $v $l $f $t $i $a
}
```

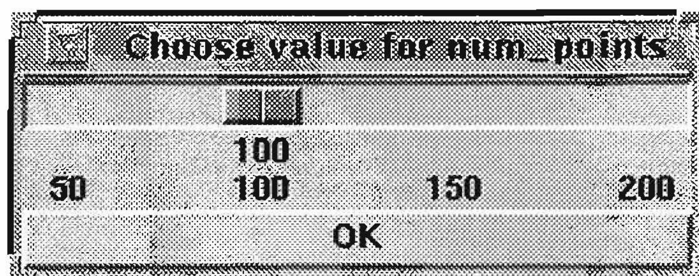


Figure 5.13. The **hscale** Script in Word and Deed

Figure 5.10 also shows the *Mult3* cell clear-wrapped, revealing its contents[10] (two *Mult2* cells) before its metamorphosis into a strictly behavioral cell. Its behavior, as shown above, exploits the associative property of multiplication to change a pair of binary multiplications into one ternary expression.

In this vistafication, the cells in the dataflow initialization section are obviously expendable, and could all be replaced by a single cell, whose behavior would be something like the following, which for simple arithmetic expressions, is much easier to grasp:

```
int num_turns, num_points;
double pi, theta_start, theta_final, theta_delta, theta, norm;

num_turns = 5;
num_points = 500;
pi = acos(-1.);
theta_start = 0.;
theta_final = 2 * pi * num_turns;
theta_delta = (theta_final - theta_start) / num_points;
theta = theta_start;
norm = theta_final;
```

Finally, for some visual/textual language expressions that are certainly deserving of a place in the documentation, if not in the actual program implementation, the x, y and z functions of *theta* computed by *ComputeData* are lucidly rendered in mathematical symbols in the following three equations, in which the role of the θ_f (*theta_final*) term as a normalizing factor (directly proportional to the *num_turns* variable) is more readily discerned:

$$
\begin{aligned}
x &= \frac{\theta}{\theta_f}\cos\theta \\
y &= \frac{\theta}{\theta_f}\sin\theta \\
z &= \frac{\theta}{\theta_f}
\end{aligned}
$$

5.2 X Synthetic Events

In this section, some familiarity with the X Window System and its protocol is assumed.[11] The following paragraph will suffice for introducing the motivation for the Vista **xse** protolib.

In applications running under X, the *event loop* is where control normally resides. The user is the main supplier of X events read and responded to in this control

[10]Note that Vista cell contents are allowed to be outside the cell bounding box.

[11]The reader is referred to [84, 85, 83] for the low-level details.

loop, which for standard GUIs are predominantly keyboard, mouse button or mouse motion events. These events are queued in the X server, which manages the hardware interfaces to keyboard, mouse and other input devices. The X client (application program) gets these queued events one by one from the server, determines the type of each event, and executes a piece of code that is conceptually attached to that type of event. This code can do arbitrary computations, or make requests of the server to either draw on the display screen, or to access information maintained by the server. The server queues up these drawing or informational client requests and in turn responds to them one by one. This response can (but need not) occur before the client code returns and the event loop regains control. When some event-triggered code causes an exit from the event loop, the application typically exits too.

Many such applications are useful computation engines in and of themselves, but because of this controlling event loop paradigm, do not support batch-mode usage. The user who wants programmatic as opposed to manual interaction must employ indirect means of feeding events to the control loop. This indirection is accommodated by the X library (Xlib), which is the lowest level of programming interface to the X protocol, by way of its *XSendEvent* primitive, the adroit use of which unfortunately requires detailed knowledge of the various X event data structures, among other things.

The contrasting high-level protocol for the Vista **xse** protolib is relatively simple, thanks in part to its incorporation of a library of C++ code already available for handling the following two non-trivial tasks:

1. event descriptor parsing, and
2. file and string-based interfacing to *XSendEvent*.

Event descriptors are character strings describing the disparate types of events supported by X. They may be stored directly as strings or one per line in a text file, from which they can be read and parsed into the corresponding internal X event data structures, suitable for passing to *XSendEvent*, given an X id of the window to which these synthetic events are to be sent. Here are several examples showing the form these event descriptors take:

```
<Btn1Down> 693 360 749 381
<Btn1Up> 693 360 749 381
<Btn3Down> 695 360 751 381
<Btn3Up> 695 360 751 381
c<key>S
c<key>D
<Enter> Normal Nonlinear True 27 0 119 1
<Leave> Normal Ancestor True 0 0 81 0
<Enter> Normal Nonlinear True 2 4 58 5
<Btn1Down> 14 9 70 10
<Btn1Up> 14 9 70 10
<Leave> Normal Nonlinear True 39 36 95 37
```

```
<key>v
<Btn3Down> 264 451 320 472
<Motion> Normal 265 451 321 472
<Motion> Normal 267 455 323 476
<Motion> Normal 268 458 324 479
<Motion> Normal 269 460 325 481
<Motion> Normal 269 461 325 482
<Btn3Up> 269 461 325 482
<key>3
c<key>M
<key>Escape
s<key>parenright
```

There is in the `xse` protolib a single portless cell type, *Send*, whose sole purpose is to send a series of keystrokes (forming its instance name) to Acme by way of the `xse` Vistafier:

```
// File: Send.h

defCell(Send, (NAME))
  xse << name;
```

Other cells could be defined, but the basic services of the *xse* class are better encapsulated in more special-purpose Vistafier-like objects, as will be shown. The `xse` Vistafier normally treats a string as a sequence of characters, each of which is to be mapped to a key event. The << operators that take a `char*` or `const char*` string argument delegate to the << (*char*) operator for each character in the string, *unless* a boolean member variable (*treat_strings_as_key_events*) is false, in which case the string is treated as the descriptor of an arbitrary event, as shown above, and the library function *parseAndSendEvents* is called instead. The `xse` << (*int*) operator toggles this boolean flag, so key events can be mixed with mouse or other kinds of events in the same *push chain* of << operators.

Some examples of interaction with `xse` will be given in the following section. It should be noted that the protocol allows for some feedback, that is, the communication is not just one-way. The >> operator, for instance, provides an interface to the Xlib primitive *XQueryPointer*, which interrogates the X server to ascertain the screen position of the mouse pointer. A call of the form below gets the current x and y mouse coordinates from the server and puts them into the supplied integer variables:

```
xse >> x >> y;
```

5.3 Unix and X United

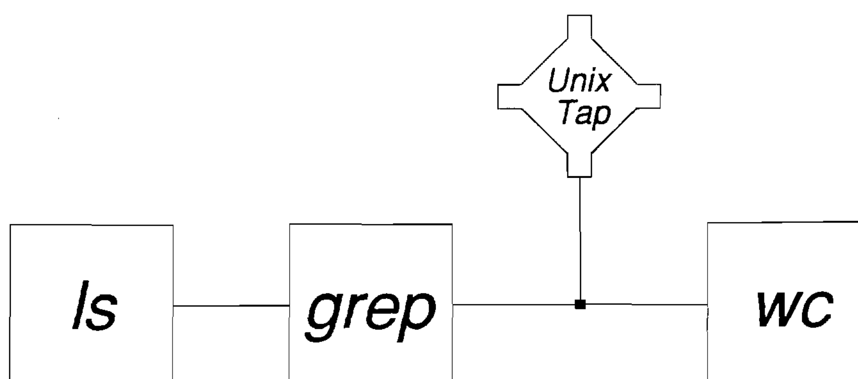
The Vista protolib discussed in this section unites the services of both **fsp** and **xse** into one **unx** Vistafier. Several new services are provided by **unx** and another pseudo-Vistafier class, **acme**, which, as its name suggests, is customized for (some) Unix and (mostly) X event interaction with Acme.

In addition, several Unix commands have been encapsulated in separate cell classes, which still use the *UnixMIO* class, but which also define modifiers to organize the various command options and parameters, rather than use the cell instance name to store a command name and arguments. That way there is a unique cell class corresponding to each Unix command, and the same command invocations can be used more than once without causing cell instance name clashes.

For example, if the pipeline in Figure 2.1 is modified to use a cell of type *UnixTap*, the resulting schematic and vistafication are shown in Figure 5.14, demonstrating among other things the concept of pipeline *tapping*, for which an alternative is pipeline *splicing*, shown in Figure 5.15.

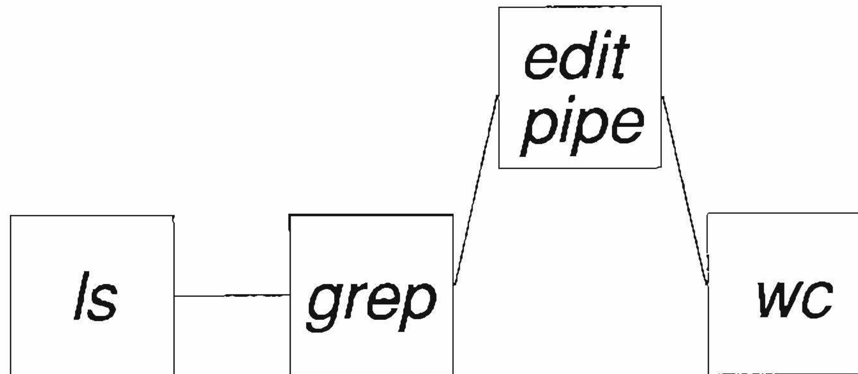
Choosing just one of these special Unix command classes, here is the definition of the *ls* cell class, to each of whose boolean options corresponds an Acme boolean modifier with its own bit weight:

```
// File: ls.h

defCell(ls, (NAME, MODS, OUTPUT(StdIO, vvout)))
  if (VISTAFIER.IsAnalyzing()) {
    UnixMIO(name, ::vvin, vvout);
  } else if (VISTAFIER.IsManifesting()) {
    UnixMIO(name, ::vvin, vvout);
  } else if (VISTAFIER.IsExecuting()) {
    var command(GetType()); static const char* boolean_options[] = {
      " -a", // list all entries
      " -c", // sort by time of last edit (or last mode change)
      " -C", // force multi-column output
      " -d", // if argument is a directory, list only its name
      " -f", // force each argument to be interpreted as a directory
      " -F", // mark directories with a trailing slash, etc.
      " -g", // show the group ownership of the file in long listing
      " -i", // for each file, print the i-number in the 1st column
      " -l", // list in long format, giving mode, owner, etc.
      " -L", // if argument is a symbolic link, list its referent
      " -r", // reverse the order of sort
      " -R", // recursively list subdirectories encountered
      " -s", // give size of each file in Kbytes
      " -t", // sort by time modified (latest first) instead of name
      " -u", // use time of last access instead of last modification
      " -1"  // force one entry per line output format
    };
```




```
// File: howmanyfiles.c

#include "howmanyfiles.h" // design
#include "unx.h" // protolib
#include "UnixTap.h" // (NAME, BIDIR(StdIO, vvl), BIDIR(StdIO, vvb),
                                BIDIR(StdIO, vvr), BIDIR(StdIO, vvt))
outclude
#include "ls.h" // (NAME, MODS, OUTPUT(StdIO, vvout))
outclude
#include "grep.h" // (NAME, MODS, INPUT(StdIO, vvin),
                                  OUTPUT(StdIO, vvout))
outclude
#include "wc.h" // (NAME, MODS, INPUT(StdIO, vvin),
                                OUTPUT(StdIO, vvout))
outclude

BEGIN

     StdIO vv2("2");
     StdIO vv1("1");

BEGIN_CELLS

    UnixTap("editpipe", ::vvl, vv2, ::vvr, ::vvt);
    ls("a", "bmods=513", vv1);
    grep("b", "bmods=256 pattern='neff'", vv1, vv2);
    wc("c", "bmods=1", vv2, ::vvout);

END_CELLS

END
```

Figure 5.14. Pipeline Tapping




```
// File: howmanyfilesagain.c

#include "howmanyfilesagain.h" // design
#include "unx.h" // protolib
#include "editpipe.h" // (NAME, INPUT(StdIO, vvin),
                                  OUTPUT(StdIO, vvout))
outclude
#include "ls.h" // (NAME, MODS, OUTPUT(StdIO, vvout))
outclude
#include "grep.h" // (NAME, MODS, INPUT(StdIO, vvin),
                                    OUTPUT(StdIO, vvout))
outclude
#include "wc.h" // (NAME, MODS, INPUT(StdIO, vvin),
                                  OUTPUT(StdIO, vvout))
outclude

BEGIN

     StdIO vv3("3");
     StdIO vv2("2");
     StdIO vv1("1");

BEGIN_CELLS

    editpipe("a", vv2, vv3);
    ls("b", "bmods=513", vv1);
    grep("c", "bmods=256 pattern='neff'", vv1, vv2);
    wc("d", "bmods=1", vv3, ::vvout);

END_CELLS

END
```

Figure 5.15. Pipeline Splicing

```
  unx.SetBooleanOptions(command, mods, boolean_options,
                        sizeof(boolean_options)/sizeof(const char*));
  command += unx.Glob(mods);
  UnixMIO(command, ::vvin, vvout);
}
```

The *unx::SetBooleanOptions* method parses the "bmods=513" mods string, unpacks its bits and appends the string corresponding to each asserted bit to the command string. Thus, the actual command seen by the executing *UnixMIO* instantiation is "`ls -a -l`" (giving a long and complete directory listing). Similarly, the *wc* cell with its "bmods=1" modifier turns into the `wc -l` command.

Recall that the purpose of this pipeline is to answer the question, "How many files (in the current directory) are mine?" The operation of tapping (or splicing) represents the afterthought, "Oh, and by the way, what are they?" The **editpipe** program is just a shell script that puts what it reads from its standard input into a temporary file, and then invokes a text editor (e.g., **xedit**) on that file. When the user exits this text editor, after possibly changing the file (pipe) contents, or saving it as a permanent file, etc., the script **cat**s the file to its standard output so that the (maybe modified) contents can continue down the pipeline.

This is an example of how to eavesdrop on a tool interaction, as was mentioned in Section 3.4 on page 30. As the user can modify the inter-tool message, it is actually more like intercepting than eavesdropping, or wire-tapping, but it is still a useful operation. When doing so is an afterthought, it is often easier to tap than to splice, as splicing may require pushing cells apart to make room for the intervening cell. Regardless, using a *UnixTap* cell to tap a pipeline is functionally equivalent to splicing in a cell, so which operation to use is a matter of preference.

5.3.1 Back Annotation

The *acme* class *EnterText* method is reproduced below to show how Acme operations can be "canned" and "vistamatically" performed via synthetic X events:

```
void EnterText(const char* text, int x, int y, boolean overwrite){
  xse << 'O'; // unselects all objects
  xse << 'N'; // name wires (or enter text label)
  xse << 0;   // now treat strings as events to parse and send
  xse << unx.ButtonEventString(1, 1, x, y);
  xse << unx.ButtonEventString(1, 0, x, y);
  if (overwrite)
    // select whole text so new entering overwrites old
    xse << "c<Key>u";
  else
    // move to end of line so new entering appends to old
    xse << "c<Key>e";
  xse << 0;   // back to strings as keyevents
  xse << text;
```

```
    xse << 0;
    xse << "<Key>Escape"; // to exit text editing mode
    xse << unx.ButtonEventString(3, 1, x, y);
    xse << unx.ButtonEventString(3, 0, x, y);
    xse << "c<Key>e"; // Not Modified
    xse << 0;
  }
```

The x and y parameters specify where in Acme's window the text is to be entered. Another *acme* method shows an encapsulated *EnterText* call, where the *SelectWarp* method first selects an Acme object by name, then warps the mouse pointer to a position centered over that object, and the *xse* >> operator is used to extract the coordinates of that position:

```
  void ShowWireValue(const char* wire_name, const char* wire_value,
                     boolean overwrite) {
    SelectWarp(wire_name);
    sleep(1); // wait a second for mouse pointer to be warped
    int x, y;
    xse >> x >> y; // gets mouse pointer position
    EnterText(wire_value, x, y+10, overwrite);
  }
```

The use of these (and a few other) methods to implement annotation of simulation results back to Acme, subsequent to interaction with the **simppl** PPL simulator, is demonstrated by the following vistafication, which drives this simple **simppl** simulation of the "canonical exor" (exclusive-or) PPL circuit:

```
// File: exor.c

#include "exor.h" // design
#include "unx.h" // protolib
#include "Set.h" // (NAME, MODS, OUTPUT(WireSpec, vvout))
outclude
#include "Get.h" // (NAME, INPUT(WireSpec, vvin))
outclude
#include "simppl.h" // (NAME, MODS, INPUT(WireSpec, vvset),
                                    OUTPUT(WireSpec, vvget))
outclude

BEGIN

    WireSpec vv1("1");
    WireSpec vv2("2");
```

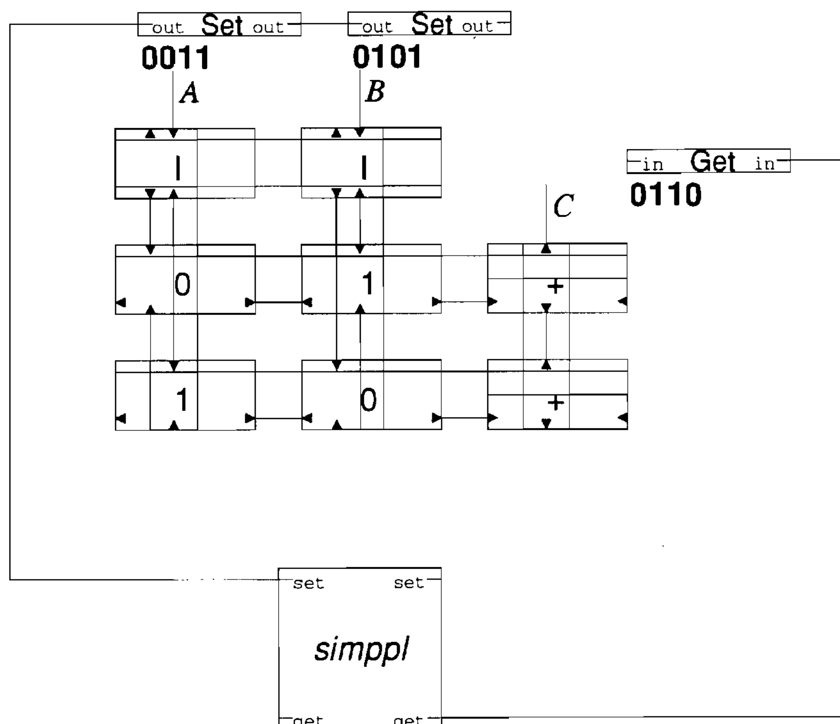


Figure 5.16. Back-Annotation from **simppl** to Acme

```
BEGIN_CELLS

    Set("A", "bmods=0 bits='0011'", vv1);
    Set("B", "bmods=0 bits='0101'", vv1);
    Get("C", vv2);
    simppl("exor", "bmods=2", vv1, vv2);

END_CELLS

END
```

Figure 5.16 shows the schematic with the back-annotated simulation values that appear during the execution of this `exor` vistafication. Note that the `exor` PPL circuit being simulated does not get vistafied, as it is in a different protolib context that is just *showing through* the **unx** `exor` context.

5.3.2 Comments to Code

Suppose management wants to know the per-programmer comments-to-code ratio for some programming project, perhaps in order to commend programmers who

comment their code adequately, and reprimand those who do not. The Vistafied agent commissioned to glean this information must carry out these nine steps:

1. generate the full pathnames of all source code files for the project,
2. identify the author of each file,
3. propagate the file contents to a comment recognizer,
4. forward these contents minus comments to **wc**,
5. feed the unfiltered code directly to **wc**,
6. read both outputs from **wc**,
7. maintain running per-author byte totals of commented code and comment-stripped code,
8. compute a final quotient of comments to code, and
9. output author, totals and quotient, suitably formatted for a management-style report.

The first step, pathname generation, is done by the so-called *globbing* operation, which takes a pathname pattern containing *wildcard* characters, and expands it by matching the pattern against the files in the specified directories. As the Unix shell already has this capability built into it, expediency dictates using the shell to *glob* these complete pathnames. The pattern for Acme is:

```
~/acme/sources/*/*/*.[ch]
```

Several simplifying assumptions are made for the second step of file author identification, so that a simple **awk** script can be employed. One of these assumptions is that each *locally-written* Acme source file contains a comment near the top of the file of the form:

```
// Author: Rick Neff
```

The reason for qualifying *locally-written* is that included among the Acme sources are several files "borrowed" from InterViews. These files have no comment of any form indicating authorship, so their author is left unidentified, although a comments-to-code ratio is still computed for them. A final assumption is that, in the not uncommon case of a single file with multiple authors, the primary author is listed first, and is the one who gets credit for all comments and code in the file.

A forked pipeline feeds these file pathnames to **awk** for step two, and for steps three through five to both **wc** and a special-purpose Vista-specified C++ comment stripper, which is the subject of the next section. The last three steps are conveniently bundled into one reasonably simple **awk** script, which assumes the three data streams have been joined into one, and sorted by author so that each author's counts appear together. Figure 5.17 contains the schematic for the following vistafied implementation of these nine steps:

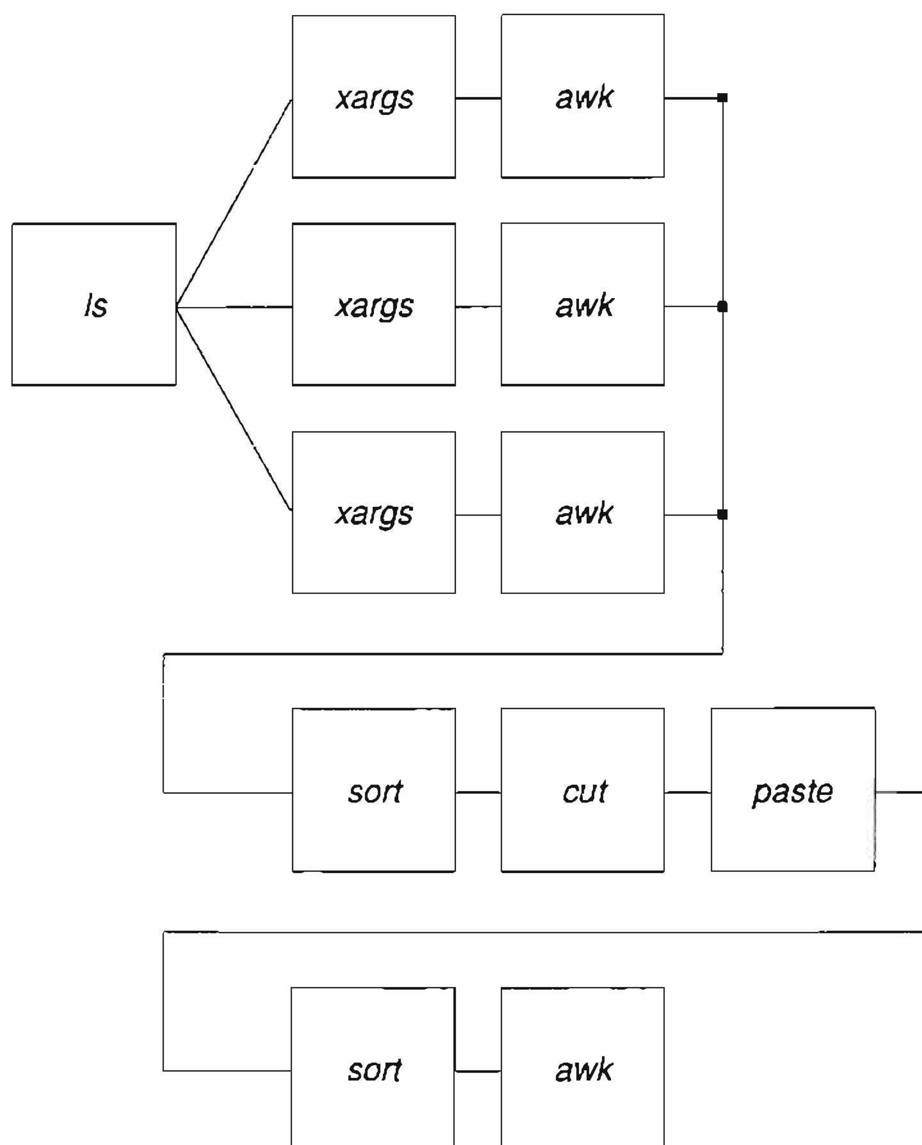


Figure 5.17. A "Looping" Multifurcated Pipeline

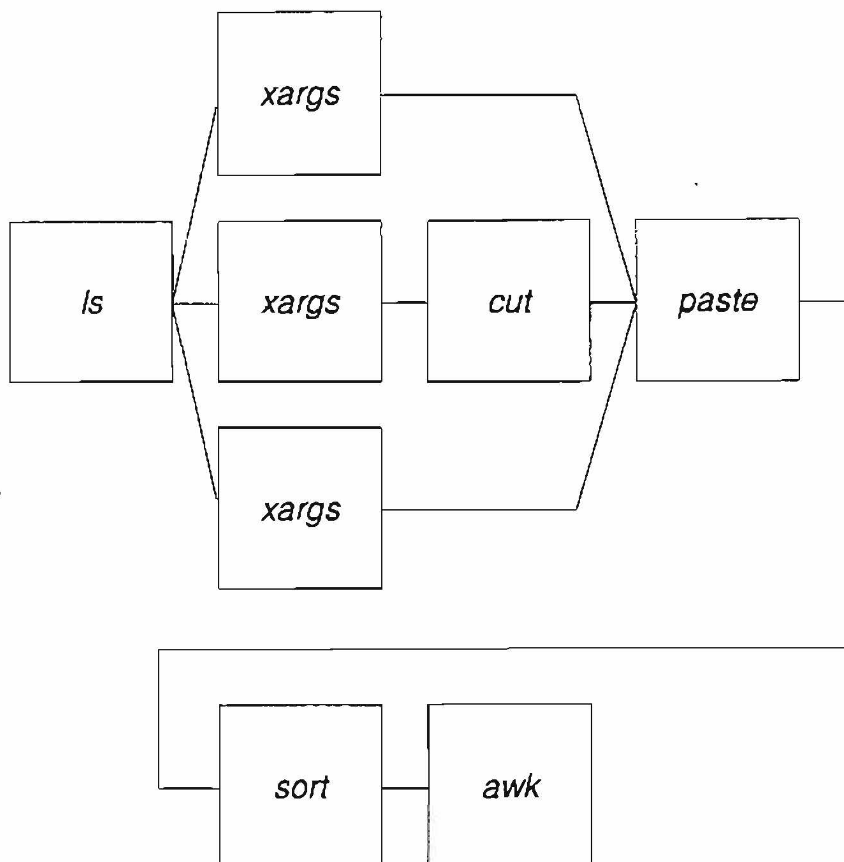
```
BEGIN

     StdIO vv1("1"); StdIO vv9("9"); StdIO vv8("8");
     StdIO vv6("6"); StdIO vv2("2"); StdIO vv5("5");
     StdIO vv4("4"); StdIO vv7("7"); StdIO vv3("3");
BEGIN_CELLS

    xargs("a", "bmods=0 numargs='1' command='awk -f auth.awk'", vv1, vv2);
    awk("b", "bmods=0 args='{print$1,10*NR+1} OFS=:'", vv2, vv3);
    ls("c", "bmods=0 glob='../sources/*/*/*.[ch]'", vv1);
    xargs("d", "bmods=0 numargs='1' command='wc -c'", vv1, vv4);
    awk("e", "bmods=0 args='{print$1,10*NR+2} OFS=:'", vv4, vv3);
    xargs("f", "bmods=0 numargs='1' command='stripper.sh'", vv1, vv5);
    awk("g", "bmods=0 args='{print$1,10*NR+3} OFS=:'", vv5, vv3);
    sort("h", "bmods=0 args='-t: +1 -n'", vv3, vv6);
    cut("i", "bmods=0 delimiter=':' fields='1'", vv6, vv7);
    paste("j", "bmods=0 args='- - -'", vv7, vv8);
    sort("k", "bmods=0", vv8, vv9);
    awk("l", "bmods=0 args='-f ct2cd.awk'", vv9, ::vvout);

END_CELLS

END
```

Note the use of the Unix **xargs** command as a loop mechanism. The **xargs** program reads arguments from its standard input, and appends them to a specified command (with fixed initial arguments), which it then executes one or more times. In this case, the supplied *numargs* string modifier value translates into the "-n1" command-line option, which specifies that only one argument should be read for each command invocation.

The **awk** cells taking the output from the three **xargs** cells are there to extract the first field from each stream output line, and tack on a *sort key*. Because the three streams produce their output at different rates, it is necessary to sort their joined stream (and then **cut** off the sort key) so that **paste** can take the linear listing of $\langle author, totalbytes, strippedbytes \rangle$ tuples, where each of the three components is on a separate line, and fold it into an output stream with one tuple per line.

A simplification of this vistafication is displayed in Figure 5.18, which demonstrates the desirability of having the **dispatch** program do *ordered joining*, where each stream being joined "waits its turn" to write to the common output stream.

An even simpler alternative can be implemented as the shell script shown below. The main advantages of this approach are that the code is pure text, and because the shell has built-in iteration constructs and file redirection, the need for **xargs** is eliminated, as is the need for the one-line "stripper.sh" shell script (`stripper < $1 | wc -c`). Additionally, as **awk** can accept its program text on the command line, all the code (both **sh** and **awk** commands) can reside in a single file. The disadvantage, of course, is that three temporary files (f1, f2 and f3) are required to store the intermediate results.




```
BEGIN

     StdIO vv3("3"); StdIO vv1("1"); StdIO vv5("5");
     StdIO vv4("4"); StdIO vv2("2");

BEGIN_CELLS

    xargs("f", "bmods=0 numargs='1' command='awk -f auth.awk'", vv1, vv2);
    ls("c", "bmods=0 glob='../sources/*/*/*.[ch]'", vv1);
    xargs("a", "bmods=0 numargs='1' command='wc -c'", vv1, vv3);
    cut("b", "bmods=0 columns='-8'", vv3, vv2);
    paste("e", "bmods=0 args='- - -'", vv2, vv4);
    xargs("d", "bmods=0 numargs='1' command='stripper.sh'", vv1, vv2);
    sort("g", "bmods=0 args='+2'", vv4, vv5);
    awk("h", "bmods=0 args='-f ct2code.awk'", vv5, ::vvout);

END_CELLS

END
```

Figure 5.18. A "Looping" Multifurcated Pipeline with Ordered Joining

```
for f in /home/grad/neff/acme/sources/*/*/*.[ch]; do
  awk 'BEGIN { A=0 }
      /Author/ {print $3; A=1; exit}
      END { if (A==0) print "????" }' $f >> f1
  wc -c < $f >> f2
  stripper < $f | wc -c >> f3
done

paste f1 f2 f3 | sort | awk '
BEGIN { printf "Author   Code+Cmnts   Code Only   Cmnt2Cd\n\n" }
{ if ($1==A) {
    T += $2; C += $3
  } else {
    if (A != "") {
      printf "%5s%12d%12d%11.2f\n", A, T, C, (T-C)/C
    }
    A=$1; T=$2; C=$3
  }
}
END { printf "%5s%12d%12d%11.2f\n", A, T, C, (T-C)/C }'

rm f1 f2 f3
```

The reason for the **cut** cell in Figure 5.18 is that **wc** outputs the filename in addition to its count(s) when invoked with a filename argument, which it does *not* (and cannot) do when reading the file from its standard input, e.g., `wc -c < $f` as in the above shell script, which therefore needs no **cut**. The output of this script, which is identical to the output of the **ct2code** vistafication, is as follows:

```
Author   Code+Cmnts   Code Only   Cmnt2Cd

 ????     574049       415350       0.38
 Brad     550956       412934       0.33
 Mike      90805        84702       0.07
 Rick    1161921       957061       0.21
 Tony     769111       668357       0.15
```

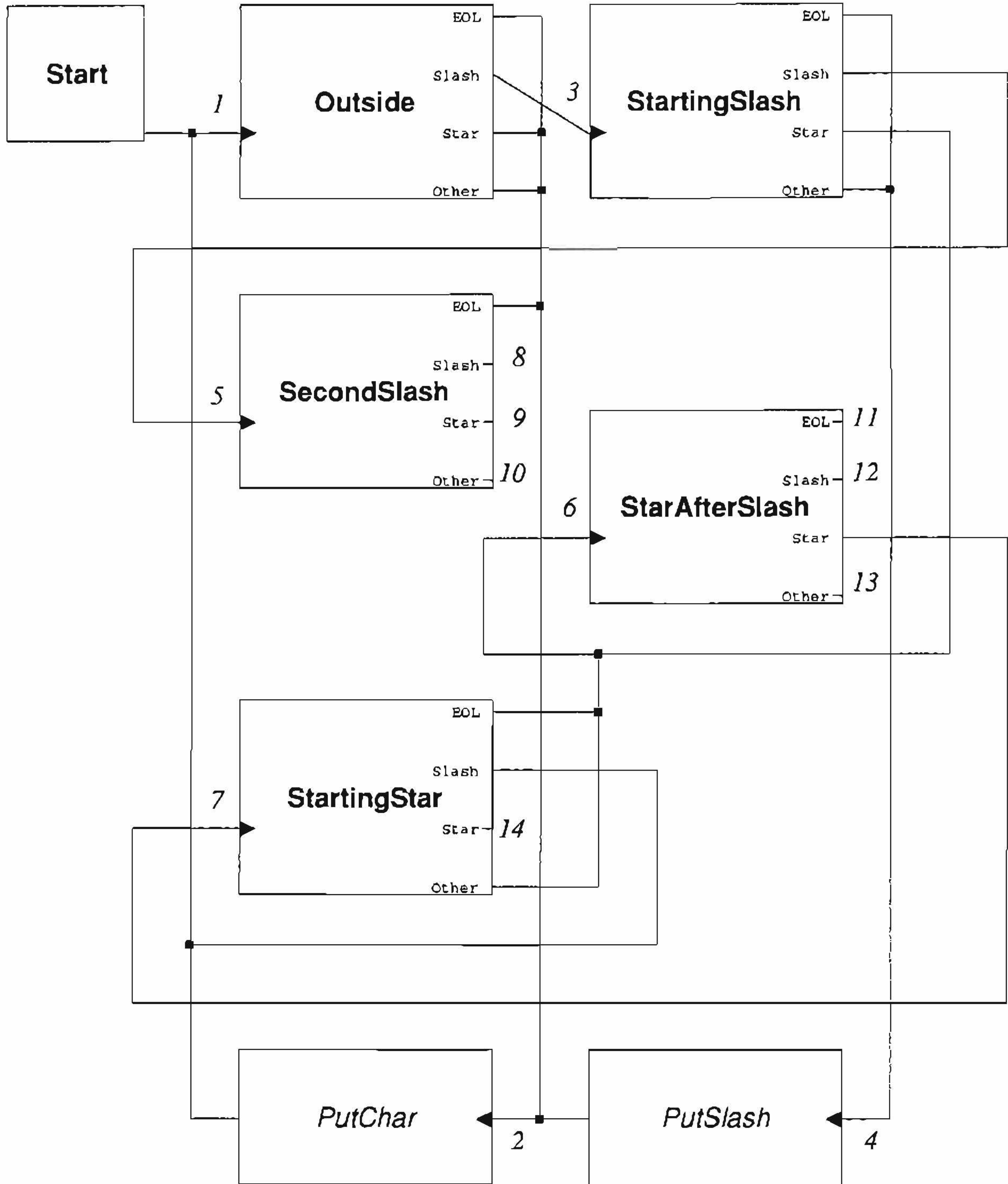


Figure 5.19. The Stripper FSM Schematic

5.4 C++ Comment Stripper State Machine

This section presents the recognizer FSM example mentioned in Section 4.5. Figure 5.19 shows the schematic for which the vistafication is as follows:

```
// File: stripper.c

#include "stripper.h" // design
#include "fsm.h" // protolib
#include "StartState.h" // (NAME, OUTPUT(Transition, vvStart))
outclude
#include "StripperState.h" // (NAME, INPUT(Transition, vvin)...
outclude
#include "StripperAction.h" // (NAME, INPUT(Transition, vvin)....
outclude

BEGIN

    Transition vv12("12"); Transition vv14("14");
    Transition vv9("9");   Transition vv3("3");
    Transition vv5("5");   Transition vv6("6");
    Transition vv1("1");   Transition vv4("4");
    Transition vv13("13"); Transition vv11("11");
    Transition vv10("10"); Transition vv8("8");
    Transition vv7("7");   Transition vv2("2");

BEGIN_CELLS

   StartState("Start", vv1);
   StripperState("Outside", vv1, vv2, vv3, vv2, vv2);
   StripperState("StartingSlash", vv3, vv4, vv5, vv6, vv4);
   StripperState("SecondSlash", vv5, vv2, vv8, vv9, vv10);
   StripperState("StarAfterSlash", vv6, vv11, vv12, vv7, vv13);
   StripperState("StartingStar", vv7, vv6, vv1, vv14, vv6);
   StripperAction("PutChar", vv2, vv1);
   StripperAction("PutSlash", vv4, vv2);

END_CELLS

END
```

For this schematic diagram, a different wrapper has been created for each cell, so that each is shown labeled with its unique instance name. The wires in the schematic are also labeled with their names. Note that wires 8-14 are tiny stub wires, which are allowed as convenient "shorthand" for specifying *self*-transitions. That is, for example, the *Star* transition from the *SecondSlash* state must re-enter that state, so wire 9 conceptually attaches to wire 5, which is the input to that state. Without this convenience, the schematic would be cluttered with several additional wire connections.

Once again, the rules state that each port must have a wire attached to it, so the Analyzer and the Manifester must catch incorrectly connected cells, and alert the user to the problem.

For the **fsm** protocol there is no need for iterated cell instantiations; however, for proper execution, four key string-valued macros must be defined in the stripper header file:

```
// File: StartState.h

defState(StartState, (NAME, OUTPUT(Transition, Start)))
  LEAVES(Start, name);



// File: StripperState.h

defState(StripperState, (NAME, INPUT(Transition, in),
                               OUTPUT(Transition, EOL),
                               OUTPUT(Transition, Slash),
                               OUTPUT(Transition, Star),
                               OUTPUT(Transition, Other)))
  ENTERS(in, name);
  LEAVES(EOL, name);
  LEAVES(Slash, name);
  LEAVES(Star, name);
  LEAVES(Other, name);



// File: StripperAction.h

defAction(StripperAction, (NAME, INPUT(Transition, in),
                                 OUTPUT(Transition, out)))
  in.EntersAction(name);
  out.LeavesAction(name);
```

Figure 5.20. Definitions of Two State and One Action Cell

```
// File: stripper.h
#define BEGIN MAIN_BEGIN
#define BEGIN_CELLS
#define END_CELLS
#define END MAIN_END
#define ActorName "Stripper"
#define ActorBase "FSMContext"
#define StateBase "State"
#define StateFile "stripperGEN.h"
```

The simple definitions of the three cell classes are shown in Figure 5.20, and the even simpler definitions of the macros `defState`, `defAction`, ENTERS and LEAVES, along with the definition of the *Transition* wire class are found in Appendix A.

When the stripper vistafication is executed, the following file is generated, which is truncated here for brevity:

```
// File: stripperGEN.h (automatically generated -- DO NOT EDIT)

class StripperState : public State {
public:
  StripperState(const char* name) {
    put_state(this, name);
  }
  virtual const char* StateName(void) const {
    return "StripperState";
  }
  virtual void EOL(Stripper& c) {
    cerr << c << "No transition from EOL" << endl;
  }
  virtual void Slash(Stripper& c) {
    cerr << c << "No transition from Slash" << endl;
  }
  virtual void Star(Stripper& c) {
    cerr << c << "No transition from Star" << endl;
  }
  virtual void Other(Stripper& c) {
    cerr << c << "No transition from Other" << endl;
  }
};

class StripperStateStart : public StripperState {
public:
  virtual const char* StateName(void) const {
    return "Start";
  }
  virtual void Start(FSMContext& c) {
    c.EnterState("Outside");
  }
} Start("Start");

class StripperStateOutside : public StripperState {
public:
  virtual const char* StateName(void) const {
    return "Outside";
  }
  virtual void EOL(Stripper& c) {
    c.EnterState("Outside"); c.PutChar();
  }
  virtual void Slash(Stripper& c) {
    c.EnterState("StartingSlash");
  }
  virtual void Star(Stripper& c) {
    c.EnterState("Outside"); c.PutChar();
  }
  virtual void Other(Stripper& c) {
```

```
    c.EnterState("Outside"); c.PutChar();
  }
} Outside("Outside");
```

The stripper FSM `.h` file is not automatically generated, but thanks to the base *FSMContext* class, and the stream abstraction, it is easy to write and easy to read, as is its corresponding `.c` file.

```
// File: stripperFSM.h

#include "vista.h" // for boolean
#include <fsm/fsm.h> // for everything else
#include "stripper.h" // for StateFile

class StripperState; // forward declaration

// Stripper is the context class of the C++ comment stripper.
// It knows about its environmental input and output streams,
// how to read and write characters, and which critical
// characters to look for in the input stream.

class Stripper : public FSMContext {
private:
  char c;
  istream& i;
  ostream& o;
  char slash, star, eol;
public:
  Stripper(istream& i, ostream& o)
    : i(i), o(o), slash('/'), star('*'), eol('\n') {
      EnterState("Start");
      Start();
  }
  ~Stripper(void) {
  }
  void PutChar(void) {
    o << c;
  }
  void PutSlash(void) {
    o << slash;
  }
  boolean GetChar(void) {
    return (i.get(c) != NULL);
  }
  StripperState& GetState(void) {
    return ((StripperState&)*itsState);
  }
  // GotChar cannot be inlined because it calls GetState to
  // invoke StripperState methods which have not yet been defined.
  void GotChar(void);
};
```

```
#include StateFile // defines StripperState and its subclasses
```

Because the main function is so short, it is included in the stripper FSM `.c` file, which also implements the `Stripper::GotChar` method that controls the `GetChar` event-driven transitions of this recognizer FSM:

```
// File: stripperFSM.c

#include "stripperFSM.h"

// GotChar directs transitions based on the type
// of character read by GetChar. The three comment
// determining characters are slash, star and eol.

void Stripper::GotChar(void) {
  if (c == slash) {
    GetState().Slash(*this);
  } else if (c == star) {
    GetState().Star(*this);
  } else if (c == eol) {
    GetState().EOL(*this);
  } else {
    GetState().Other(*this);
  }
}

int main(void) {
  Stripper stripper(cin, cout);
  while (stripper.GetChar()) {
    stripper.GotChar();
  }
  return 0;
}
```

5.5 Sockets and RPC

Below the level of Unix pipes[12] are the interprocess communication abstractions known as *sockets*, which act as endpoints for communication in a specified *domain*, for example, the local Unix domain (for processes running on the same Unix host), and the ARPA Internet domain (for those running on different but internetworked hosts). There are three types of sockets, here briefly described[13] with their chief characteristics:

- *stream* socket—bidirectional byte-stream data, guaranteed to be reliable, sequenced and unduplicated, supported by the (TCP) *Transmission Control Protocol*.
- *datagram* socket—bidirectional data packets, *not* guaranteed to be transmitted reliably, sequenced or unduplicated, supported by the (UDP) *User Datagram Protocol*.
- *raw* socket—gives access to primitive network protocols, supported by the (IP) *Internet Protocol*.

To effectively use sockets one must master the following system calls: **accept**, **bind**, **connect**, **listen**, **select** and **socket**, among others, which govern the establishment and interplay of socket connections. Alternatively, one may seek, hopefully find, and over time, learn to use a library of code that adds one or more abstraction layers to these low-level functions. The InterViews *Dispatch* library is such a suite of classes providing these shielding layers. Table 5.1 displays the names of this library's base classes with both one-line and short-paragraph synoptic excerpts from [52]. Those flagged with a dagger are *abstract* interface classes, not meant to be directly instantiated. The user is responsible for deriving from them suitable subclasses, and fleshing out their virtual methods or adding other methods as required by the protocol.

For example, an *RpcReader* (which itself inherits from *IOHandler*, which was mentioned in Section 5.1) needs a derived class to define the action to be taken when the socket connection is closed by the client, by overriding the virtual *connectionClosed* method. More importantly, this derived class must define static member functions to unmarshall RPC requests, and initialize a member function array with their addresses. Each of these RPC-request-unmarshalling functions must be designed to extract any needed arguments from the *rpcstream* representing the connection, execute the request, and insert into the *rpcstream* any return values to be sent back to the client.

Part of the Vista commission is removing impediments to tool (library) use. High and shielding though their abstraction level may be, these RPC classes still present a challenge to their would-be users. Vista, via its **cts** (client to server) and **ptp** (peer to peer) protolib protocols, exemplified below, takes these class concepts

---

[12]Pipes are in fact just a special case of Unix-domain sockets, and are typically implemented with the **socketpair** system call, which creates a pair of connected sockets.

[13]See [101] (Chapter 10) for a good general introduction to sockets.

Name	Synopsis
iostreamb	*unformatted streams*
	istreamb, ostreamb, and iostreamb are streams just like istream, ostream, and iostream except for two features. First, they automatically insert and extract delimiters around datums where necessary so one never has to manually separate datums by whitespace. Second, they can insert and extract integers as either unformatted (binary) bytes or formatted characters so as to eliminate the time needed to convert integers to strings and back. Both of these features make these classes easier to use than the base stream classes for inter-process communication (IPC).
rpcstream	*iostreamb specialized to RPC requests*
	rpcstream, irpcstream, and orpcstream specialize iostreamb, istreamb, and ostreamb, respectively, to RPC requests. That is, the associated streambuf will be an rpcbuf.
rpcbuf	*streambuf specialized for sending and receiving RPC requests*
	An rpcbuf is a streambuf specialized in two ways: to use an IPC connection as a source or sink of characters and to send and receive RPC requests. The rpcbuf encloses RPC requests in packets which begin with a length field so as to buffer requests until they are complete if using non-blocking I/O.
RpcReader†	*read RPC requests from a client*
	An RpcReader reads RPC requests from an rpcstream which represents a connection to a client. When it reads an RPC request, it uses the request number to look up the address of a function in an array and calls that function to unmarshall the request's arguments and execute the request.
RpcWriter†	*write RPC requests to a server*
	An RpcWriter writes RPC requests to a server. Derived classes should add member functions that send RPC requests corresponding to the RPC service's protocol.
RpcHdr	*header for remote procedure calls*
	An RpcHdr enables an RPC request to be sent or received. To send an RPC request, one would insert an RpcHdr into an rpcstream followed by any arguments and then flush the rpcstream if one wanted the request to be sent immediately. The rpcstream automatically fills in each RPC request's length field. To receive an RPC request, one would extract an RpcHdr from an rpcstream and examine the "request()" member to determine which additional arguments need to be extracted as well.
RpcService†	*support RPC between a service and its clients*
	An RpcService creates a port and listens to it for connections from clients. When a client opens a connection, the RpcService will create a reader to handle RPC requests from the connection.
RpcPeer†	*support bidirectional RPC between two services*
	An RpcPeer tries to open a connection to another RpcPeer. If the other RpcPeer is not yet running, the RpcPeer will create its own port and wait for the other RpcPeer to open a connection. When either RpcPeer opens a connection, each RpcPeer will create both a reader and a writer so each RpcPeer can send RPC requests to its opposite over the same connection.

Table 5.1. InterViews Dispatch Library RPC Classes

still higher, and clears the RPC way for the non-expert programmer/user through automatic generation of the necessary but tedious-to-write "glue" and "stub" code.

5.5.1 Client to Server

The example of a Client/Server RPC interaction presented here is a moderately useful enhancement of the **xse** program, the source code for which provided the library mentioned above in Section 5.2 for use by the **xse** protolib. The name chosen for the **cts** vistafication of this interaction is **sxe** (pronounced es-ex-ee). In the spirit of the **fsm** protolib, the execution of the **sxe** vistafication, shown below, causes the generation of its two split personalities, those being the **sxeclient** and the **sxeserver** executables.

```
BEGIN

    ExitMessage vv4("4");
    WindowEventsStream vv2("2");
    WindowEventsFile vv1("1");
    Medium vv_sxe_(".sxe.");
    WindowEventsArray vv3("3");

BEGIN_CELLS

    Client("client", vv_sxe_);
    SendWindowEventsFile("swef", vv_sxe_, vv1);
    ParseFileSendEvents("pfse", vv1, vv_sxe_);
    Server("server", vv_sxe_);
    SendWindowEventsStream("swes", vv_sxe_, vv2);
    ParseStreamSendEvents("psse", vv2, vv_sxe_);
    SendWindowEventsArray("swea", vv_sxe_, vv3);
    ParseArraySendEvents("pase", vv3, vv_sxe_);
    StopRunning("sr", vv_sxe_, vv4);
    Shutdown("sd", vv4, vv_sxe_);

END_CELLS

END
```

Figure 5.21 shows the seemingly symmetrical **sxe** schematic, which in fact *is* symmetrical topologically, but not behaviorally, as a look at the definitions of the ten cell classes will reveal. While at first glance there appear to be more than five wires, on clear-wrapping it is evident in Figure 5.22 that the internally-connected equivalent ports of both the *Client* and the *Server* cells merge the top wire (labeled ".sxe.") into one wire that fully and explicitly interconnects all ten cells. This wire is of type *Medium*. The wires labeled 1-4 that tie the eight so-called *push/pull pairs* of cells together are of four different types. By simply attaching more such pairs in

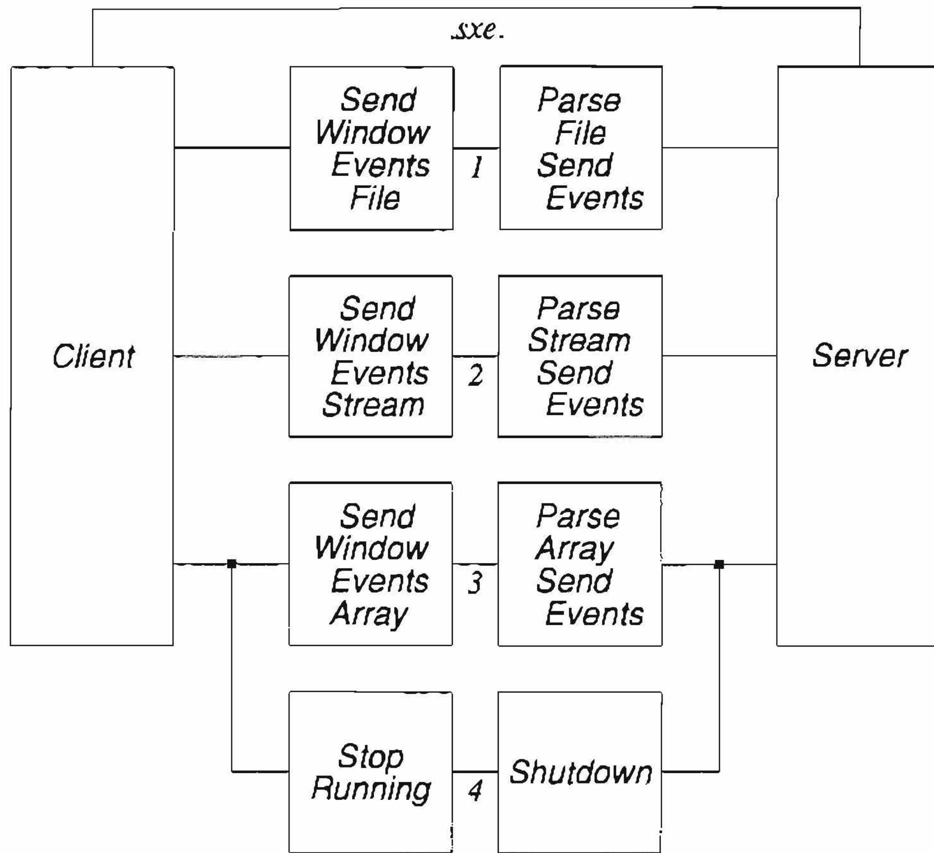


Figure 5.21. The sxe Client/Server Schematic

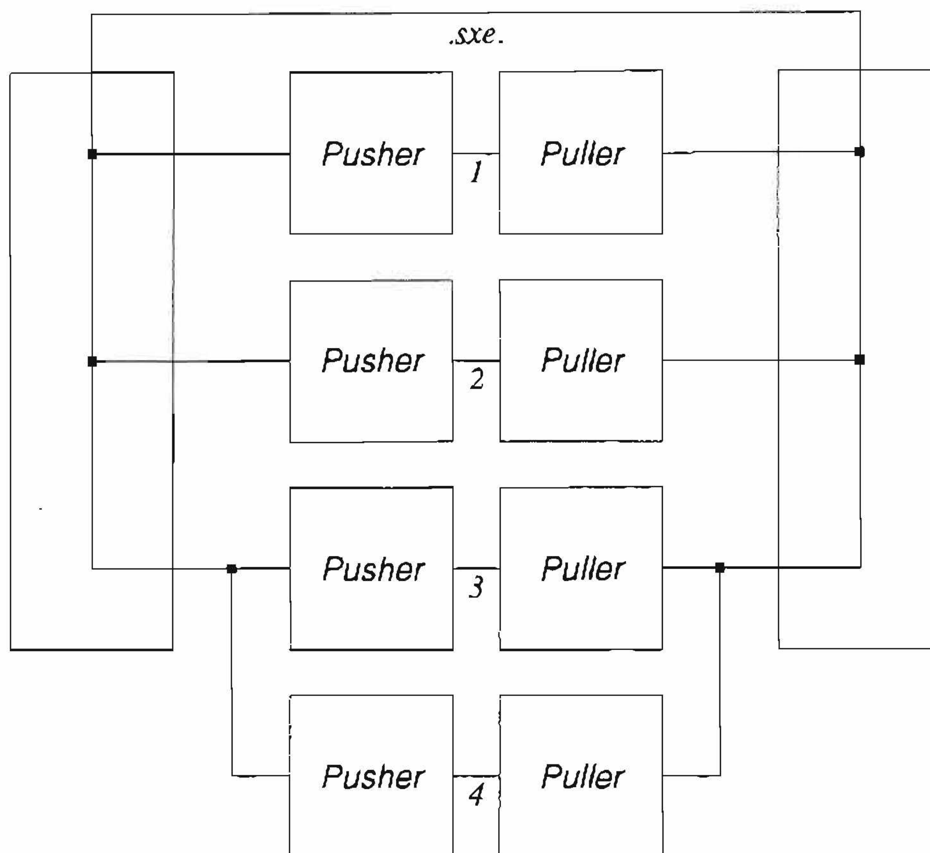


Figure 5.22. Fully-Interconnected Push/Pull Pairs

like manner as the bottom pair, the number of different client/server interactions can be extended as desired. In fact, the bottom push/pull pair was added to provide a graceful means of shutting down the **sxeserver** by having the **sxeclient** tell it when to stop running. The other three pairs, as their names also suggest, are for handling three mutually-exclusive client request styles. Thus, **sxeclient** sends a window identifier and event descriptors to **sxeserver** in one of these three ways, itemized below with an example of their use:

- *File*: `sxeclient 'Workview 4.1.0 060591' workview-quit.sxe`
- *Stream*: `echo ^Xzsxe.c | xsekey | sxeclient 'Minibuffer @ acme'`
- *Array*: `sxeclient 0x1300002 'Mod1<Btn3Down>' '<key>Return'`

The main service provided by **sxeserver** is to parse the event descriptors and send their corresponding synthesized X events to the specified window. Note that the intended window recipient can be identified by either name or X id. Another time-saving service supplied by **sxeserver** is to maintain a mapping of window name to window id, a convenience *not* provided by the X server.

The class names of the client-side *Pusher* cell, the *Medium* wire and the server-side *Puller* cell are of the form $SendWindowEvents\langle thing\rangle$, $WindowEvents\langle thing\rangle$

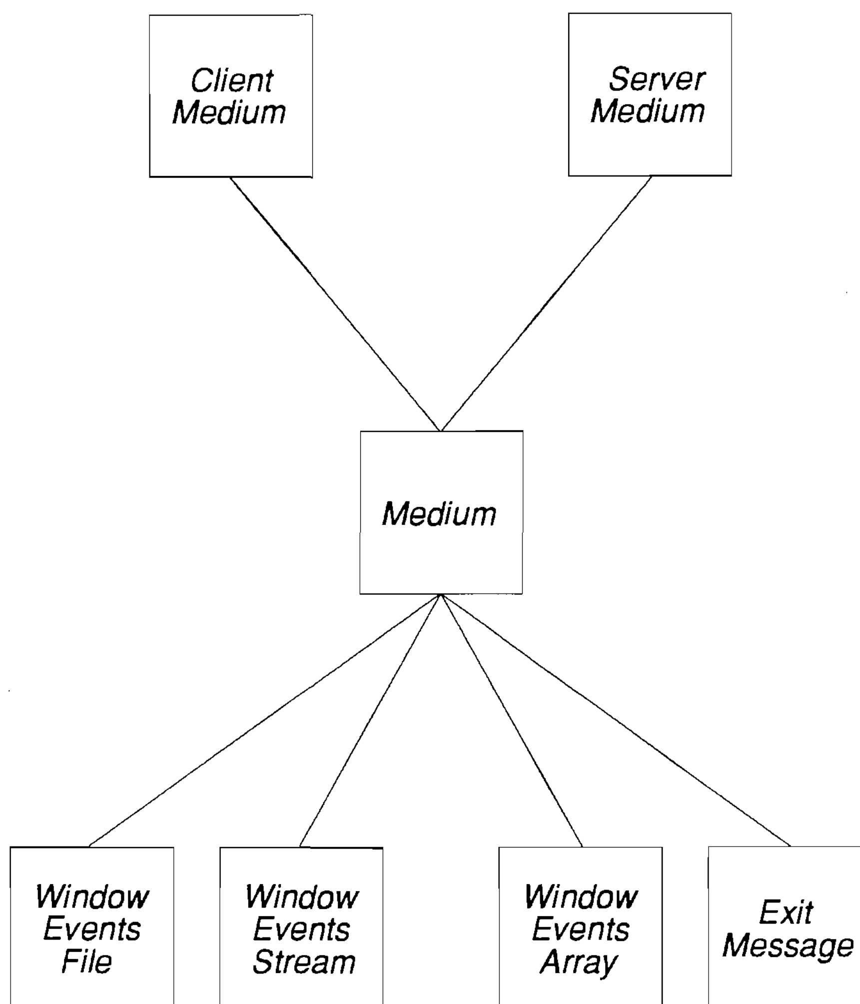


Figure 5.23. The **sxe** Wire Type Inheritance Hierarchy

and *Parse⟨thing⟩SendEvents*, respectively, where ⟨*thing*⟩ is either *File*, *Stream* or *Array*. Note per Figure 5.22 that there is a kind of *inheritance by containment* happening here. Each client-side cell must be created by wrapping up a single *Pusher* cell instance and defining two ports. The type of the port on the left must be *ClientMedium* and the type of the port on the right must be a *sub*type of *Medium*. Similarly, a server-side cell prototype contains a *Puller* cell instance, with the same *Medium* subtyped port on the left, and a *ServerMedium*-type port on the right. These two wire types, *ClientMedium* and *ServerMedium*, are not subtypes but rather *super*types of *Medium*. Figure 5.23 illustrates the inheritance hierarchy of these wire types, from which the Analyzer (conceptually) does its type inferencing while vistafying. What these wires and cells do is shown in Figure 5.24 and Figure 5.25 for just the stream case, the others being similar.

```
// File: WindowEventsStream.h

class WindowEventsStream : public Medium {
public:
  char* window_id; char cur_event[80];
  WindowEventsStream(const char* name)
    : Medium("WindowEventsStream", name), window_id(nil) {
    cur_event[0] = '\0';
  }
  WindowEventsStream(int n) {
    window_id = new char[n];
  }
  ~WindowEventsStream(void) {
    delete [] window_id;
  }
  boolean GetNextEvent(istream& is) {
    if (is.good()) {
      is.getline(cur_event, 79);
      return true;
    } else {
      return false;
    }
  }
  boolean IsEmpty(void) {
    return (cur_event[0] == '\0');
  }
  boolean IsGood(void) {
    if (VISTAFIER.IsDone() && VISTAFIER.argc == 2) {
      window_id = strdup(VISTAFIER.argv[1]);
      return true;
    } else {
      return false;
    }
  }
};
```

Figure 5.24. The *WindowEventsStream* Class Definition

```
// File: SendWindowEventsStream.h

#include "WindowEventsStream.h"

#define BEGIN_SendWindowEventsStream_CELLS VISTAFIER.EnterPusher(self);

#define END_SendWindowEventsStream_CELLS

defCell(SendWindowEventsStream, (NAME,
        BIDIR(ClientMedium, vvClientMedium),
        BIDIR(WindowEventsStream, vvWindowEventsStream)))
  if (vvClientMedium.IsGood(&vvWindowEventsStream)) {
    vvClientMedium.GetServer() << vvWindowEventsStream.window_id
                               << check;
    while (vvWindowEventsStream.GetNextEvent(cin)) {
      vvClientMedium.GetServer() << vvWindowEventsStream.cur_event;
    }
  }


// File: ParseStreamSendEvents.h

#include "WindowEventsStream.h"
#include <xse/event.h>

#define BEGIN_ParseStreamSendEvents_CELLS VISTAFIER.EnterPuller(self);

#define END_ParseStreamSendEvents_CELLS

defCell(ParseStreamSendEvents, (NAME,
        BIDIR(WindowEventsStream, vvWindowEventsStream),
        BIDIR(ServerMedium, vvServerMedium)))
  if (vvServerMedium.IsReady()) {
    vvServerMedium.GetClient() >> vvWindowEventsStream.window_id;
    for (;;) {
      vvServerMedium.GetClient() >> vvWindowEventsStream.cur_event;
      if (vvWindowEventsStream.IsEmpty()) {
        break;
      }
      parseAndSendEventsFromStream(vvWindowEventsStream.window_id,
                                   vvWindowEventsStream.cur_event);
    }
  }
```

Figure 5.25. Two "sxe" Pusher/Puller Cell Definitions

The *SendWindowEventsStream* cell calls its *ClientMedium* wire's *IsGood* method to determine (by examination of the Vistafier's command-line arguments) if it is the one (of four cells) that should be operational. A side effect of this method, if it returns true, is to copy the command-line window id into its own `window_id` data member. If the medium "is good" then the window id is pushed to the server by invoking the *rpcstream* << operator, after which the stream is checked to see if it is still good (no end-of-file or failure status bits set). If so, then while there are event-descriptor data to be read from the standard input stream (`cin`) the *GetNextEvent* method returns true, and one by one, the event descriptors (stored temporarily in `cur_event`) are likewise pushed to the server. In a similar loop, the server pulls each datum from the client, parses and sends an event to the window (whose id was the first thing the server pulled that the client pushed).

There is more going on behind the scenes in the *IsGood* method (see Appendix A), but it should be evident that the preponderance of functionality resides not in the client, but as is usually the case, in the server. The `<xse/event.h>` file `#included` above declares, among others, the `parseAndSendEventsFromStream` function, and the *xse* library where these functions are defined is linked (only) into the `sxeserver` executable.

The *Parse⟨thing⟩SendEvents* cells and their corresponding wires are used to implement the static member functions needed for the *automatically* defined *Reader* subclass of *RpcReader*, as shown:

```
// File: sxereader.h (automatically generated -- DO NOT EDIT)

#define CELL3 ParseFileSendEvents
#define WIRE3 WindowEventsFile

#define CELL2 ParseStreamSendEvents
#define WIRE2 WindowEventsStream

#define CELL4 ParseArraySendEvents
#define WIRE4 WindowEventsArray

#define CELL1 Shutdown
#define WIRE1 ExitMessage

#define NUMFN 4

#define READ(N) \
  static void read##N(RpcReader* reader, RpcHdr& hdr,rpcstream& client){\
    Medium mdm("read"#N,(Reader*)reader, &client); \
    WIRE##N arg(hdr.ndata()); \
    CELL##N("read"#N, arg, mdm); \
  }

class Reader : public RpcReader {
protected:
  RpcService* _service;
  virtual void connectionClosed(int fd) {
    close(fd);
```

```
    if (Service::no_longer_needed) {
      _service->quitRunning();
    }
    delete this;
  }
  READ(1)
  READ(2)
  READ(3)
  READ(4)
public:
  Reader(int fd, RpcService* service)
    : RpcReader(fd, NUMFN + 1, /* binary = */ true),
    _service(service) {
    client().setf(ios::dont_close);
    _function[1] = &Reader::read1;
    _function[2] = &Reader::read2;
    _function[3] = &Reader::read3;
    _function[4] = &Reader::read4;
  }
  virtual ~Reader(void) {
  }
};

void Service::createReader(int fd) {
  new Reader(fd, this);
}
```

Expanding the macros for the second function shows more clearly the protocol followed by each *reader* function, which is simply to instantiate the two wires and the single cell involved in this client-request-reading operation.

```
static void read2(RpcReader* reader, RpcHdr& hdr, rpcstream& client) {
  Medium mdm("read2", (Reader*)reader, &client);
  WindowEventsStream arg(hdr.ndata());
  ParseStreamSendEvents("read2", arg, mdm);
}
```

Note that the `ndata()` member function of *RpcHdr* is used by the special integer-taking *WindowEventsStream* wire constructor. Thus, this wire can allocate the storage needed to hold the window id, which is put there by the *ParseStream-SendEvents* cell when it pulls this data from the client by invoking the *rpcstream* >> operator:

```
vvServerMedium.GetClient() >> vvWindowEventsStream.window_id;
```

This only happens if the *ServerMedium::IsReady* method returns true, which it does if the sole instance of the *Service* subclass of *RpcService* is running, which it is after the original cells in the "serverized" vistafication are instantiated, as can be seen by the **sxeserver**'s header file, which is reproduced in Figure 5.26, along with its similar but simpler "clientized" counterpart.

```
// File: sxeserver.h (automatically generated -- DO NOT EDIT)

#define BEGIN MAIN_BEGIN
#define BEGIN_CELLS
#define END_CELLS service->Run();
#define END MAIN_END

#define SERVER

// File: sxeclient.h (automatically generated -- DO NOT EDIT)

#define BEGIN MAIN_BEGIN
#define BEGIN_CELLS
#define END_CELLS
#define END MAIN_END

#define CLIENT
```

Figure 5.26. Automatically Generated Client/Server Headers

Also generated by purely textual transformations of the original `sxe.h` and `sxe.c` files, in conjunction with the *output* of the executing **sxe** vistafication, the corresponding `.c` files for **sxeclient** and **sxeserver** appear in Figures 5.27 and 5.28. Note that this code is essentially identical to what would result if the schematic in Figure 5.21 were cut exactly in half vertically, and each half were vistafied separately.

5.5.2 Peer to Peer

The **ptp** protolib borrows a great deal of protocol from **cts**, as an interaction between peers is similar (but naturally not identical) to a client-server interaction. The vistafication described below implements a simple RPC exchange of an integer between two processes *running the same program*. The code for this program, **exipeer**, is derived from the vistafied **exi** code in much the same fashion as **sxeclient** and **sxeserver** were cloned from **sxe**. In this case a single program is called for, as there is no clear separation of functionality, no real distinction between who is serving and who is being served—both peers serve each other.

Figure 5.29 shows how much easier a peer to peer connection is made than is a client/server one. Furthermore, while not infeasible it is entirely unnecessary to "cleave" or "unzip" this vistafication to produce the corresponding **exipeer** code:

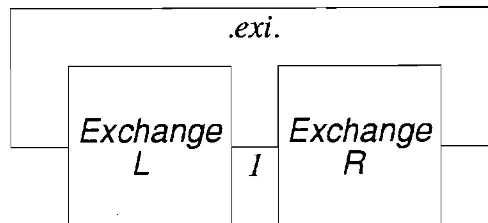
```
// File: exipeer.h (automatically generated -- DO NOT EDIT)

#define BEGIN MAIN_BEGIN
```

```
// File: sxeclient.c (automatically generated -- DO NOT EDIT)

#include "sxeclient.h"
#include "cts.h"

#include "Client.h"
outclude
#include "SendWindowEventsFile.h"
outclude
#include "SendWindowEventsStream.h"
outclude
#include "SendWindowEventsArray.h"
outclude
#include "StopRunning.h"
outclude

BEGIN

     Medium vv0(".sxe.");
     ExitMessage vv4("4");
     WindowEventsStream vv2("2");
     WindowEventsFile vv1("1");
     WindowEventsArray vv3("3");

BEGIN_CELLS

    Client("client", vv0);
    SendWindowEventsFile("swef", vv0, vv1);
    SendWindowEventsStream("swes", vv0, vv2);
    SendWindowEventsArray("swea", vv0, vv3);
    StopRunning("sr", vv0, vv4);

END_CELLS

END
```

Figure 5.27. Automatically Generated Client Code

```
// File: sxeserver.c (automatically generated -- DO NOT EDIT)

#include "sxeserver.h"
#include "cts.h"

#include "Server.h"
outclude
#include "ParseFileSendEvents.h"
outclude
#include "ParseStreamSendEvents.h"
outclude
#include "ParseArraySendEvents.h"
outclude
#include "Shutdown.h"
outclude

#include "sxereader.h"

BEGIN

     Medium vv0(".sxe.");
     ExitMessage vv4("4");
     WindowEventsStream vv2("2");
     WindowEventsFile vv1("1");
     WindowEventsArray vv3("3");

BEGIN_CELLS

    Server("server", vv0);
    ParseFileSendEvents("pfse", vv1, vv0);
    ParseStreamSendEvents("psse", vv2, vv0);
    ParseArraySendEvents("pase", vv3, vv0);
    Shutdown("sd", vv4, vv0);

END_CELLS

END
```

Figure 5.28. Automatically Generated Server Code




```
// File: exi.c

#include "exi.h" // design
#include "ptp.h" // protolib
#include "ExchangeL.h" // (NAME,
                           BIDIR(Medium, vvMedium),
                           BIDIR(Integer, vvInteger))
outclude
#include "ExchangeR.h" // (NAME,
                           BIDIR(Integer, vvInteger),
                           BIDIR(Medium, vvMedium))
outclude

BEGIN

     Integer vv1("1");
     Medium vv_exi_(".exi.");

BEGIN_CELLS

    ExchangeL("l", vv_exi_, vv1);
    ExchangeR("r", vv1, vv_exi_);

END_CELLS

END
```

Figure 5.29. The exi Peer to Peer Schematic and Vistafication

```
#define BEGIN_CELLS do {

#define END_CELLS } while (vv0.PeerShouldRun());

#define END MAIN_END

#define PEER
```

```
// File: exipeer.c (automatically generated -- DO NOT EDIT)

#include "exipeer.h" // design
#include "ptp.h" // protolib
#include "ExchangeL.h" // (NAME,
                              BIDIR(Medium, vvMedium),
                              BIDIR(Integer, vvInteger))
outclude
#include "ExchangeR.h" // (NAME,
                              BIDIR(Integer, vvInteger),
                              BIDIR(Medium, vvMedium))
outclude

#include "exireader.h"

BEGIN

     Integer vv1("1");
     Medium vv0(".exi.");

BEGIN_CELLS

    ExchangeL("l", vv0, vv1);
    ExchangeR("r", vv1, vv0);

END_CELLS

END
```

The control structure for this program is effected by having BEGIN_CELLS and END_CELLS implement a *do while Peer Should Run* loop, within which the two cells are alternately instantiated until conditions allow the loop to exit, which conditions are described below.

Like the `sxereader.h` file, the `exireader.h` file defines a *Reader* subclass of *RpcReader* as well as its static member functions, of which there is only one in this example. Also implemented in this file, unlike in the `sxereader.h` file, are the destructor and the two *createReaderAndWriter* methods of the *Peer* subclass of the *RpcPeer* class. These three *Peer* class methods cannot be defined in the **ptp** protolib `.h` file, since they reference *Reader* methods that are not defined until the `exireader.h` file is created:

```
// File: exireader.h (automatically generated -- DO NOT EDIT)

#define CELL1 ExchangeR
#define WIRE1 Integer

#define NUMFN 1

#define READ(N) \
  static void read##N(RpcReader* reader, \
                      RpcHdr& hdr,rpcstream& client){\
    Medium mdm("read"#N,(Reader*)reader, &client); \
    WIRE##N arg(hdr.ndata()); \
    CELL##N("read"#N, arg, mdm); \
  }

class Reader : public RpcReader {
protected:
  Peer* peer;
  virtual void connectionClosed(int fd) {
  }
  READ(1)
public:
  Reader(rpcstream* client, Peer* peer)
    : RpcReader(client, NUMFN + 1),
    peer(peer) {
    client->setf(ios::dont_close);
    _function[1] = &Reader::read1;
  }
};

virtual Peer::~Peer(void) {
  delete reader;
  delete writer;
}

virtual boolean Peer::createReaderAndWriter
                (const char* rHost, int rPort) {
  writer = new Writer(rHost, rPort);
  if (writer->server()) {
    reader = new Reader(&writer->server(), this);
    return true;
  } else {
    Error(rHost, rPort);
    return false;
  }
}

virtual void Peer::createReaderAndWriter(int fd) {
  if (writer) {
    Error(fd, 0);
    return;
  }
```

```
    writer = new Writer(fd);
    if (writer->server()) {
      reader = new Reader(&writer->server(), this);
    } else {
      Error(fd);
    }
  }
```

Again for clarity, the sole *reader* function is shown here expanded:

```
static void read1(RpcReader* reader,
                  RpcHdr& hdr, rpcstream& client) {
  Medium mdm("read1", (Reader*)reader, &client);
  Integer arg(hdr.ndata());
  ExchangeR("read1", arg, mdm);
}
```

The class definitions of the *Integer* type *Medium* wire, and the complementary "left" (local) *ExchangeL* and "right" (remote) *ExchangeR* cells are as follows:

```
class Integer : public Medium {
public:
  int datum;
  boolean got_it;
  Integer(const char* name) : Medium("Integer", name),
    datum(0), got_it(false) {
  }
  Integer(int) : datum(0), got_it(true) {
  }
  ~Integer(void) {
    if (got_it) {
      cout << datum << endl;
    }
  }
  boolean IsGood(void) {
    if (VISTAFIER.IsDone() && VISTAFIER.argc == 2) {
      datum = atoi(VISTAFIER.argv[1]);
      return true;
    } else {
      return false;
    }
  }
};

defCell(ExchangeL, (NAME,
                    BIDIR(Medium, vvMedium),
                    BIDIR(Integer, vvInteger)))
  if (vvMedium.IsGood(&vvInteger)) {
    vvMedium.GetStream() << vvInteger.datum << flush;
    vvMedium.Done();
  }
```

```
defCell(ExchangeR, (NAME,
                    BIDIR(Integer, vvInteger),
                    BIDIR(Medium, vvMedium)))
  if (vvMedium.IsReady()) {
    vvMedium.GetStream() >> vvInteger.datum;
    vvMedium.Done();
  }
```

As in the client-server protocol, the *Medium* still needs to be "good and ready" to effect an RPC data exchange. Both cells have a chance to pass the goodness and readiness tests, and they succeed in that order in both peer processes, regardless of which process starts executing first. Each peer process plays both roles—*local* (or left) and *remote* (or right). The local peer's *ExchangeL* cell, on passing the goodness test, pushes its integer to its remote peer. Then it pretends to be the remote peer, and waits to pull the integer that will be (or already has been) pushed by its counterpart, which it does when its *ExchangeR* cell passes the readiness test. It is an asymmetry in the **ptp** protocol that the *ExchangeR* cell instantiation where the *Medium* is ready is *not* the instantiation that occurs in the main loop. Rather, it is the instantiation occurring when the *Reader::read1* static method shown above is called, as the only *Medium* wire that is ready to have its *rpcstream* "client" member read from is the one that is instantiated with an *rpcstream* argument. Note that the *Integer* wire instantiated in *Reader::read1* has a non-cleanup type of task for its destructor to do, which it does surreptitiously before *read1* returns; namely, writing the remotely-received integer datum to the standard output.

The normal *rpcstream* push and pull operators are used to send and receive the integer. Once the local peer has pushed its integer and pulled (and printed out) the integer pushed by the remote peer, having nothing else to do, it invokes the *Medium::Done* method, which asserts two static boolean flags in the *Peer* object, `local_done` the first time and `remote_done` the second time. These two flags conditionalize the *Peer::ShouldRun* method, which in turn controls the main loop (as seen above):

```
boolean ShouldRun(void) {
  if (!local_done || !remote_done) {
    Dispatcher::instance().dispatch();
  }
  return (!local_done || !remote_done);
}
```

The *Dispatcher::dispatch* method is the same one used by the **dispatch** program (see Section 5.1 above) to encapsulate the low-level *select* system call, which in this case is synchronously multiplexing socket I/O instead of pipe I/O. To end this cursory exposition of the **ptp** protocol, here is a quick peek at the number of function-call layers between *select* and the *ExchangeR* cell instantiation:

1. *select* is called by
2. *Dispatcher::dispatch*, which (when *select* returns) calls

3. *Dispatcher::notify*, which calls
4. *RpcReader::inputReady*, which calls
5. *RpcReader::execute*, which calls the static member function
6. *Reader::read1*, which instantiates (calls)
7. *Integer* and *ExchangeR*.

In the following section is presented a more sophisticated example of a peer-to-peer interaction.

5.6 A Tale of Two Spreadsheets

Spreadsheets have attained their immense popularity because of their friendly, intuitive interface and their enormous flexibility[49, 39, 1]. State-of-the-art spreadsheet programs (e.g., Lotus 123) go as far as providing "addin" toolkits that allow one to write C language routines to manipulate the spreadsheet, perform customized functions, and so forth. There is no direct way, however, to make one Unix spreadsheet talk to another one—one using the other as a computation engine.

There is, of course, an indirect way. In the vistafication described in this section, two Unix spreadsheet programs, **sc** and **oleo**, communicate via an agent that knows how to talk to each, and can translate from the command language of one to that of the other. The **ptp** Reader/Writer RPC protocol is once again employed, and as this provides for network communication, the two spreadsheets are run on different hosts to exercise this feature. Other highlights of this vistafication include:

1. parameterization by polymorphism, and
2. merging behavior by multiple inheritance.

Specifically crafted to perform this spreadsheet experiment, the **sse** protolib references **fsp** and **xse** in addition to the just described **ptp** protolib. The *sse* Vistafier class acquires the *IOHandler* protocol by (multiple) inheritance, and uses polymorphic delegation to a *SpreadSheet* class object to parameterize by type of spreadsheet, as explained further below. The **tts** vistafication, shown in Figures 5.30 and 5.31, uses the same main control structure (loop while peer should run) as did **exi** above. The generated `ttsreader.h` is likewise similar to the `exireader.h` file, except that it defines two *reader* functions instead of just one. The purpose of these two functions is to pull integer and string data pushed from the peer, exchanging the integer datum in much the same fashion as in **exipeer** above. Thus, there are two RPC cell-wire-cell handshake trios connected by a common *Medium*; namely, $\langle Xrow \leftrightarrow RowCol \leftrightarrow Xcol \rangle$ and $\langle Send \rightarrow String \rightarrow Recv \rangle$. The code for the former trio appears in Figure 5.32, and in Figure 5.33 for the latter.

The cell-quartet of *Choose*, *Put*, *Get* and *Quit*, whose behavior is shown in Figure 5.34, implements a very simple command interpreter that neutralizes the vagaries of each spreadsheet's command syntax. Table 5.2 gives a sample comparison of command and expression syntax for **sc** and **oleo**, both with each other and

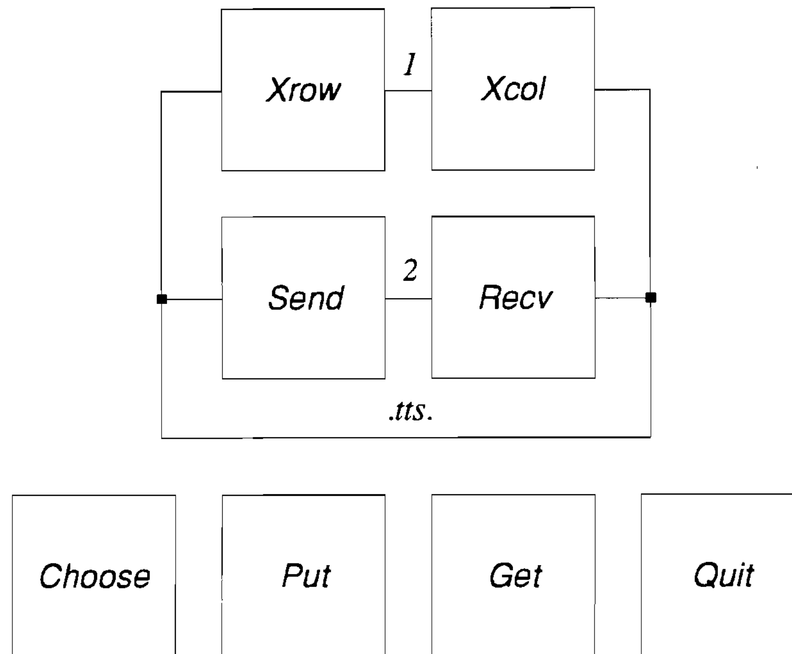


Figure 5.30. The tts Spreadsheet Schematic

with the neutral syntax recognized by these cells when they are instantiated in the main control loop of the executing **ttspeer** agent.

The **ttspeer** agent relies on short shell scripts to invoke the two spreadsheets. Each script starts its spreadsheet running in the background, and returns the id of the X window in which the spreadsheet is running, so that commands can be communicated by having the *xse* Vistafier send synthetic X events from the agent to each spreadsheet. Here is the text of the two executable scripts, named **sc** and **oleo** for reasons made known below:

```
#! /bin/csh -f
set name = 'Spreadsheet Calculator'
toolwait xterm -T "$name" -geometry 80x24+0+0 \
               -xrm "*allowSendEvents:true" \
               -e ~/spreadsheets/sc-6.21/sc
xwininfo -name "$name" -int | grep xwininfo | cut -d" " -f5

#! /bin/csh -f
set name = 'Oleo version 1.5'
toolwait ~/spreadsheets/oleo-1.5/oleo
xwininfo -name "$name" -int | grep xwininfo | cut -d" " -f5
```

A utility program that controls X client startup, **toolwait** takes the name of an X client program as an argument, and executes it. Only after the client program has started will **toolwait** exit. Thus, when started immediately thereafter, **xwininfo**, another utility that finds and displays information about X windows, will find a

```
// File: tts.c

#include "tts.h" // design
#include "sse.h" // protolib
#include "Xrow.h" // (NAME, BIDIR(Medium, vvMedium),
                             BIDIR(RowCol, vvRowCol))
outclude
#include "Xcol.h" // (NAME, BIDIR(RowCol, vvRowCol),
                             BIDIR(Medium, vvMedium))
outclude
#include "Send.h" // (NAME, BIDIR(Medium, vvMedium),
                             BIDIR(String, vvString))
outclude
#include "Recv.h" // (NAME, BIDIR(String, vvString),
                             BIDIR(Medium, vvMedium))
outclude
#include "Choose.h" // (NAME)
outclude
#include "Put.h" // (NAME)
outclude
#include "Get.h" // (NAME)
outclude
#include "Quit.h" // (NAME)
outclude

BEGIN

     RowCol vv1("1");
     String vv2("2");
     Medium vv_tts_(".tts.");

BEGIN_CELLS

    Xrow("xrow", vv_tts_, vv1);
    Xcol("xcol", vv1, vv_tts_);
    Send("send", vv_tts_, vv2);
    Recv("recv", vv2, vv_tts_);
    Choose("choose");
    Put("put");
    Get("get");
    Quit("quit");

END_CELLS

END
```

Figure 5.31. The tts Spreadsheet Vistafication

```
// File: RowCol.h

class RowCol : public Medium {
public:
  int datum;
  boolean got_it;
  RowCol(const char* name) : Medium("RowCol", name),
    datum(0), got_it(false) {
  }
  RowCol(int) : datum(0), got_it(true) {
  }
  ~RowCol(void) {
    if (got_it) {
      VISTAFIER.spreadsheet->SetRow(datum);
    }
  }
  boolean IsGood(void) {
    if (VISTAFIER.GotCommand("xrowcol")) {
      datum = VISTAFIER.spreadsheet->name.length();
      VISTAFIER.spreadsheet->SetCol(datum);
      return true;
    } else {
      return false;
    }
  }
};

// File: Xrow.h

#include "RowCol.h"

defCell(Xrow, (NAME, BIDIR(Medium, vvMedium),
                     BIDIR(RowCol, vvRowCol)))
  if (vvMedium.IsGood(&vvRowCol)) {
    vvMedium.GetStream() << vvRowCol.datum << flush;
  }

// File: Xcol.h

#include "RowCol.h"

defCell(Xcol, (NAME, BIDIR(RowCol, vvRowCol),
                     BIDIR(Medium, vvMedium)))
  if (vvMedium.IsReady()) {
    vvMedium.GetStream() >> vvRowCol.datum;
  }
```

Figure 5.32. A Bidirectional RPC Integer Exchange

```
class String : public Medium {
public:
  char* datum;
  boolean got_it;
  String(const char* name) : Medium("String", name),
    datum(nil), got_it(false) {
  }
  String(int n) : got_it(true) {
    datum = new char[n];
  }
  ~String(void) {
    if (got_it) {
      int c, r;
      VISTAFIER.spreadsheet->GetColRow(c, r);
      VISTAFIER.spreadsheet->SelectCell(r, c);
      VISTAFIER.spreadsheet->PutStringValue(datum);
    }
    delete [] datum;
  }
  boolean IsGood(void) {
    if (VISTAFIER.GotCommand("send")) {
      int c, r;
      VISTAFIER.spreadsheet->GetColRow(c, r);
      VISTAFIER.spreadsheet->SetCurrentRegion(c,r,c+7,r);
      datum = VISTAFIER.spreadsheet->GetStringValue();
      return true;
    } else {
      return false;
    }
  }
};

defCell(Send, (NAME, BIDIR(Medium, vvMedium),
                     BIDIR(String, vvString)))
  if (vvMedium.IsGood(&vvString)) {
    vvMedium.GetStream() << vvString.datum << flush;
  }

defCell(Recv, (NAME, BIDIR(String, vvString),
                     BIDIR(Medium, vvMedium)))
  if (vvMedium.IsReady()) {
    vvMedium.GetStream() >> vvString.datum;
  }
```

Figure 5.33. A Unidirectional RPC String Delivery

```
defCell(Choose, (NAME))
  if (VISTAFIER.GotCommand(name)) {
    char type; cin >> type;
    switch (type) {
      case 'c': { // cell
        int col, row; cin >> col >> row;
        VISTAFIER.spreadsheet->SelectCell(col, row);         break; }
      case 'r': { // region
        int l, b, r, t; cin >> l >> b >> r >> t;
        VISTAFIER.spreadsheet->SetCurrentRegion(l, b, r, t); break; }
    }
  }
defCell(Put, (NAME))
  if (VISTAFIER.GotCommand(name)) {
    char type; cin >> type >> ws;
    switch (type) {
      case 'n': { // number
        double d; cin >> d;
        VISTAFIER.spreadsheet->PutNumericValue(d);   break; }
      case 's': { // string
        char s[80]; cin.getline(s, sizeof(s), '\n');
        VISTAFIER.spreadsheet->PutStringValue(s);    break; }
      case 'f': { // formula
        char f[80]; cin.getline(f, sizeof(f), '\n');
        VISTAFIER.spreadsheet->PutNumericFormula(f); break; }
      case 'l': { // literal formula
        char f[80]; cin.getline(f, sizeof(f), '\n');
        VISTAFIER.spreadsheet->PutStringFormula(f);  break; }
    }
  }
defCell(Get, (NAME))
  if (VISTAFIER.GotCommand(name)) {
    char type; cin >> type;
    switch (type) {
      case 'n': { // number
        cout << VISTAFIER.spreadsheet->GetNumericValue() << endl; break; }
      case 's': { // string
        cout << VISTAFIER.spreadsheet->GetStringValue()  << endl; break; }
    }
  }
defCell(Quit, (NAME))
  if (VISTAFIER.GotCommand(name)) {
    VISTAFIER.Quit();
  }
```

Figure 5.34. A Quartet of Cells Interpreting Spreadsheet Commands

What\Which	sc	ttspeer	oleo
		Addressing	
Cells	`b3`	`c c 2 4`[a]	`r4c2`
Regions	`a0:d2`	`c r 1 3 4 1`[b]	`r1:3c1:4`
		Value Entry	
Number	`=123`	`p n 123`[c]	`123`
String	`<abc`	`p s abc`[d]	`"abc"`
		Formula Entry	
Numeric	`=@sqrt(b3)`	`p f sqrt`[e]	`sqrt(r4c2)`
String	`ER@substr("abc",1,2)`	`p l` *expr*[f]	`substr(1,2,"abc")`
		Miscellany	
Write	`W fn a0:d2`	none[g]	`^[^P a r1:3c1:4 fn`[h]
Quit	`q n`	`q`[i]	`^X^C yes`

[a]Choose Cell at column 2, row 4.

[b]Choose Region between column 1, row 3 and column 4, row 1.

[c]Put Number 123 into the currently chosen cell.

[d]Put String "abc" into the currently chosen cell.

[e]Put Formula *name* and append the currently chosen *degenerate* region enclosed in parentheses, e.g., sqrt(a3). The command "c r 1 4 1 4" specifies a degenerate region consisting of a single cell, which collapses to the "a3" cell address.

[f]Put Literal Formula (*expr* is a literal string in the command syntax of the targeted spreadsheet).

[g]No direct command. *WriteAndReadRegion* is called by the *Get* cell to communicate string or numeric *values* computed by the spreadsheet to the agent.

[h]The generated filename *fn* comes from an instance of the *TemporaryInputFile* class (defined in the **fsp** protolib), which object is instantiated by the *WriteAndReadRegion* method.

[i]Each program asks for confirmation differently when told to quit without first saving a modified spreadsheet file.

Table 5.2. Spreadsheet Command Comparison

window with the given name and return, among other things, its id. The **grep** and **cut** pipeline is necessary to filter these "other things" out.

The reason for the **xterm** in the first script is that **sc** only runs in a terminal (or terminal emulator) program, whereas **oleo** knows how to open its own X window in which to run. Setting the "allowSendEvents" X resource to be "true" is necessary, otherwise **xterm** will silently ignore any synthetic X events sent to it.[14]

The directories where are found the actual spreadsheet executables are purposely *excluded* from the list of directories the shell searches to find executable programs. The directories where the scripts themselves reside are purposely *included* in this search path. That way there is no conflict between script and spreadsheet, even though they have the same name, and the spreadsheet program is always invoked indirectly through its corresponding script.

The *sse::Init* method looks at the first argument given on the **ttspeer** command line, treats it as the name of an executable Unix program and opens an input pipe (*ipipe*) to the program thus named. It reads through this pipe the X window id written on exit by this program, and passes the id to the *xse::Init* method.

The program thus named and executed must be either the **sc** or **oleo** script, because the *sse::Init* method also looks at this name as a string, and if the string is "sc" or "oleo" then it instantiates a new object of class *Sc* or *Oleo* and caches a pointer to this instance in its *spreadsheet* member variable, which is of type (pointer to) *Spreadsheet*. *Spreadsheet*, of course, is the parent class of both *Sc* and *Oleo*, whose virtual methods furnish the polymorphic access to each spreadsheet's idiosyncratic user interface.

The last task performed by the *sse::Init* method is the following, which it does so that the *Dispatcher::dispatch* method called in the main control loop can multiplex input between the RPC socket conduit and the standard (file descriptor 0) input channel:

```
Dispatcher::instance().link(0, Dispatcher::ReadMask, this);
```

When *select* returns indicating the availability of data from standard input, the `Dispatcher::instance()` calls the virtual *IOHandler::inputReady* method redefined in the *sse* class as follows:

```
int inputReady(int fd) {
  cin >> cmd;
  return 0;
}
```

The *cmd* member is of type *char*, and serves to store the primary command character the user types in, against which each interpreter cell compares its name

---

[14]A bit of state is set in the X event data structure by the *XSendEvent* function, providing the way for X clients to distinguish synthetic events from real ones. Allowing synthetic events to be sent to arbitrary windows (by anyone who can connect to the X server) poses a significant security risk.

to decide if it should extract further data from standard input and execute its command. Returning zero to the dispatcher instructs it to call *select* again to check the status of the file descriptor before invoking *IOHandler::inputReady* again.

Figure 5.35 depicts a sample execution of **ttspeer sc** in the bottom-left window running an **xterm** on host **vlsi**, and **ttspeer oleo** likewise running in the bottom-right window on the **acme** host. The circled-number annotations added to the captured screen image show the order in which commands were typed to these two windows. For this example, the confirmation-on-exit reply has been excised from each spreadsheet's *Quit* method, so that exiting **ttspeer** will not quite quit its spreadsheet. Thus may be seen the "save-data" prompt that each program displays, as well as the sign-off message indicating that the **ttspeer** has indeed exited. Here follows an explanation of the 22 commands in the order given (the "L:" prefix indicates input to **ttspeer sc**, and "R:" to **ttspeer oleo**):

1. L: Transmit the integer 2 (the length of "sc") as the column of the cell of **oleo** where data will be sent. Set 2 to be the column, and receive 4 from peer to be the row of the cell whose string value will be sent to peer by subsequent *send* commands. Prepare to receive data from peer in cell at column 4, row 2.
2. R: Transmit the integer 4 (the length of "oleo") as the column of the cell of **sc** where data will be sent. Set 4 to be the column, and receive 2 from peer to be the row of the cell whose string value will be sent to peer by subsequent *send* commands. Prepare to receive data from peer in cell at column 2, row 4.
3. R: Choose (and move cursor to) cell at column 4, row 2.
4. R: Put the number 123 into that cell.
5. R: Send as a string the *displayed* representation of that cell to **sc**.
6. L: Choose (and move cursor to) cell at column 1, row 2.
7. L: Choose degenerate region at column 4, row 2 to be the argument to the next command.
8. L: Put formula "ston" (string to number) in cell "a1" to convert the string received from **oleo** into its numeric equivalent.
9. L: Choose (and go to) cell at column 2, row 3.
10. L: Put the string "dollars" there.
11. L: Choose (and go to) cell at column 2, row 4.
12. L: Put the literal formula "ext(b2,a1)" there[15].

[15]Short for *external*, *ext* specifies an external function to be called to get the cell's value. It takes a string argument naming a Unix command, and a numeric argument, which is converted to a string and appended to the name argument to form a Unix command line. Using **popen** to create a pipe and a process, sc reads through the pipe the first line of output produced by the process running this Unix command, returning it as the string value of this formula. This is a useful (albeit expensive) way to get an approximation to the "addin" capability in Lotus 123. Closer and better still is the capability of **oleo** to dynamically link in object code while it is running. The drawback, of course, is that it is much more difficult to provide object code that

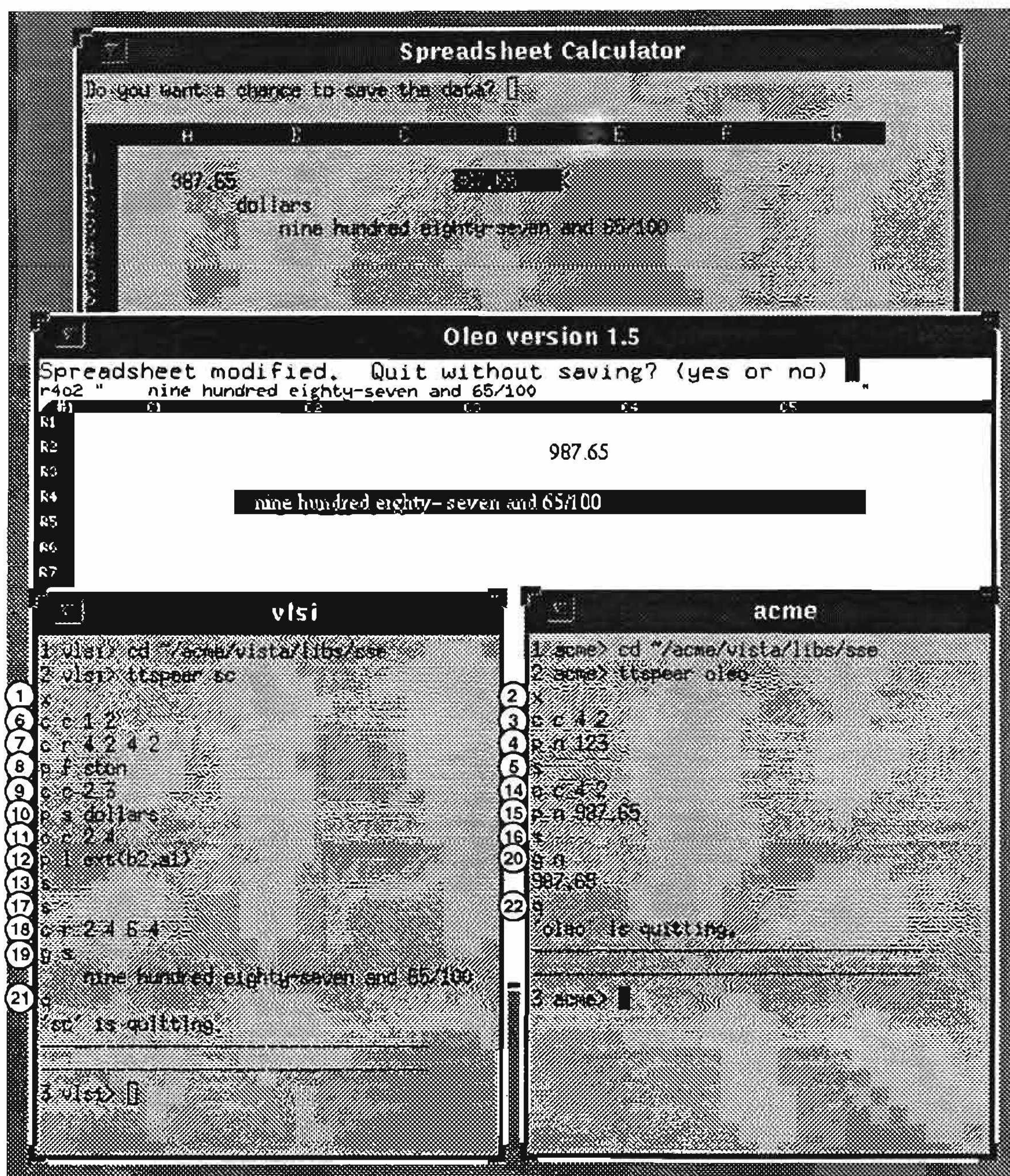


Figure 5.35. The "ttspeer" Agent in Action

13. L: Send the string value[16] in this cell back to **oleo**.
14. R: Choose (and go to) cell at column 4, row 2.
15. R: Put the number 987.65 there.
16. R: Send it back to **sc**.
17. L: Send the recomputed "dollars" equivalent back to **oleo**.
18. L: Choose region bounded by columns 2 and 6 on row 4.
19. L: Get the string displayed there from **sc** and write it to standard output. The region had to be made wider in the previous step to accommodate this label that extends beyond its cell's normal width.
20. R: Get the number displayed in the current region by **oleo** and write it to standard output.
21. L: Quit.
22. R: Quit.

It should be noted that these two spreadsheet programs are not really in the same league with state-of-the-art spreadsheet applications running under DOS or Windows on PCs. They have no GUI, no fancy icons and no menus—commands *must* be entered from the keyboard. Still, they do have the considerable advantage of being free,[17] and hence, many programmers continue to contribute to their development, to the general benefit of all who wish to use them as an alternative to high-priced commercial software.

5.7 Mollifying Egocentric Tools

Speaking of high-priced commercial software, Viewlogic Systems, Inc., which is highly representative of its kind, is a vendor of tools for designing and simulating integrated circuits. The **powerview** program provides the current controlling interface to all of Viewlogic's tools, and the strong presumption made by Viewlogic is that every user will naturally prefer to use **powerview** to launch and interact with these tools. Moreover, **powerview** assumes its environment is structured in a certain way, and forces the user to conform to this structure. The egocentric **powerview** view is imposed on users, which imposition is tolerable if Viewlogic's tools are the only ones being used. In the common situation where other tools also enter the picture, e.g., Acme, the imposition is more bothersome. The interaction

---

must be structured in a rigid way, conforming to the scheme **oleo** requires in order to load new functions and/or new keyboard commands.

[16]The result of running the command **dollars 123**, which is *one hundred twenty-three and no/100*, a string suitable for printing on a payroll check, for example.

[17]More precisely, **sc** is in the public domain and **oleo** is "copylefted" (distributed under the terms of the Free Software Foundation's GNU General Public License, which essentially is an interdiction of *any* restriction on its redistribution).

agent[18] presented in this section constitutes a mollification of the egocentric rigidity of Viewlogic's tools.

Partly because of new software releases, and partly due to a desire to make it work well, several iterations were required to design the **adsim** agent, whose purpose is to enable a **powerview** tyro to successfully accomplish the following three objectives:

1. Use the **madsnet** tool to generate a mixed analog/digital signal netlist.
2. Use the **madssim** tool to do a mixed analog/digital signal simulation.
3. Examine the simulated digital and analog waveforms using **viewtrace**.

The **adsim** agent is a shell script that delegates all the work to the Unix (actually GNU) **make** program. The `adsim.mk` makefile read by **make** is where all the functionality resides. In this makefile database are 19 targets and pseudo-targets that implicitly specify the steps to be taken to simulate a circuit containing both analog and digital components. Automated by **make** are a dozen or more tool-interaction tasks that otherwise would have to be performed manually by the user. These tasks include (for an Acme circuit design named `test` from which the user has already generated a `test.ppl` file) the following:

- Invoke the PPL tool **spplice** to produce a `test.1` file (Viewlogic's netlist format) from the `test.ppl` file.
- Move or copy the `test.1` and `test.cmd` files[19] into the `wir` subdirectory of the test project directory, using **powerview** to create this project directory (and its various subdirectories) first if it does not already exist.
- Connect to the test project directory.
- Invoke **madsnet** to create a `test.cir` **hspice** netlist file and a `test.vsm` **viewsim** netlist file.
- Edit the `test.cir` file to add the necessary analog stimulus and probe statements, and to change various hardwired voltage and time values appropriately.
- Invoke **madssim** to simulate the analog `test.cir` using **hspice** and the digital `test.vsm` using **viewsim**. (The IPC handshaking necessary for **hspice** and **viewsim** to communicate back and forth is handled by **madssim**.)
- Start and read into **viewtrace** both the `test.wfm` **viewsim** output file and the `test.tr0` **hspice** output file, to view the mixed waveforms for this simulation.

To give just a hint of how these manual steps are automated by Vista, here is an excerpt from the `adsim.mk` makefile showing the `$(PROJDIR)` target, and the rules (commands) needed to create it, followed by a number of explanatory comments:

---

[18]There are actually two—**dsim** and **adsim**. The former is a subset of the latter.

[19]If the user has provided a `test.src` file of **simppl** commands, **spplice** translates it into a `test.cmd` file containing the equivalent **viewsim** commands.

```
PVNAME  = 'Powerview Cockpit'
PVDIR   = $(shell echo $(WDIR) | cut -d: -f1)
PROJDIR = $(PVDIR)/$(NAME)

$(PROJDIR):
	@echo 'Mod1<key>x' > powerview-create-project.xse
	@echo "pdc $(PROJDIR)" | xsekey >> powerview-create-project.xse
	toolwait powerview > /dev/null
	xse -window $(PVNAME) -file powerview-create-project.xse
	until [ -d $(PROJDIR)/wir ]; do sleep 1; done
	xse -window $(PVNAME) 'Mod1<Btn3Down>' '<key>Return'
```

1. Parameterizing the project creation target are three internal **make** variables (`PVNAME`, `PVDIR` and `PROJDIR`), one external command-line-specified variable (`NAME`, whose value is "test" in this case), and one environment variable (`WDIR`). The `WDIR` variable is used in a shell interaction that extracts the first pathname from the value of this variable, which value is a colon-separated list of *working directories* that **powerview** searches to find projects and libraries. The project directory will be a subdirectory named `test` of this primary working directory.
2. The `Mod1<key>x` key event descriptor translates to the "Alt-X" keystroke chord, whose effect on **powerview** is to pop up its "Command Line" text entry box, which Viewlogic considerately[20] provides as an alternative to invoking commands solely via menus.
3. "pdc" is the **powerview** abbreviation for its "project (viewdraw) create" command. The abbreviated form is used so as to minimize the number of synthetic key events sent to **powerview**.
4. **xsekey** reads its standard input, and for every character read writes on its standard output an X key event descriptor (e.g., `s<key>x` for a shifted 'X' character), one per line.
5. **toolwait** starts **powerview** in the background, and returns when **powerview** is ready to accept input. The noisy banter **powerview** normally puts out is diverted to the `/dev/null` *bit bucket*.
6. **xse** finds the **powerview** window by name, then parses the X key events described in the just-created `powerview-create-project.xse` file and sends them to the **powerview** window.
7. The purpose of the `until` shell command is to synchronize the **powerview** and **make** processes that are running concurrently. The reason they have to be running concurrently is that **powerview** has no batch mode,[21] hence, it

[20] *Inconsiderately* however, Viewlogic is inexplicably inconsistent in making the *space bar* the key to press to get this command entry box in all its other tools.

[21] The omission of batch mode capability is another annoying artifact of egocentric interactive software.

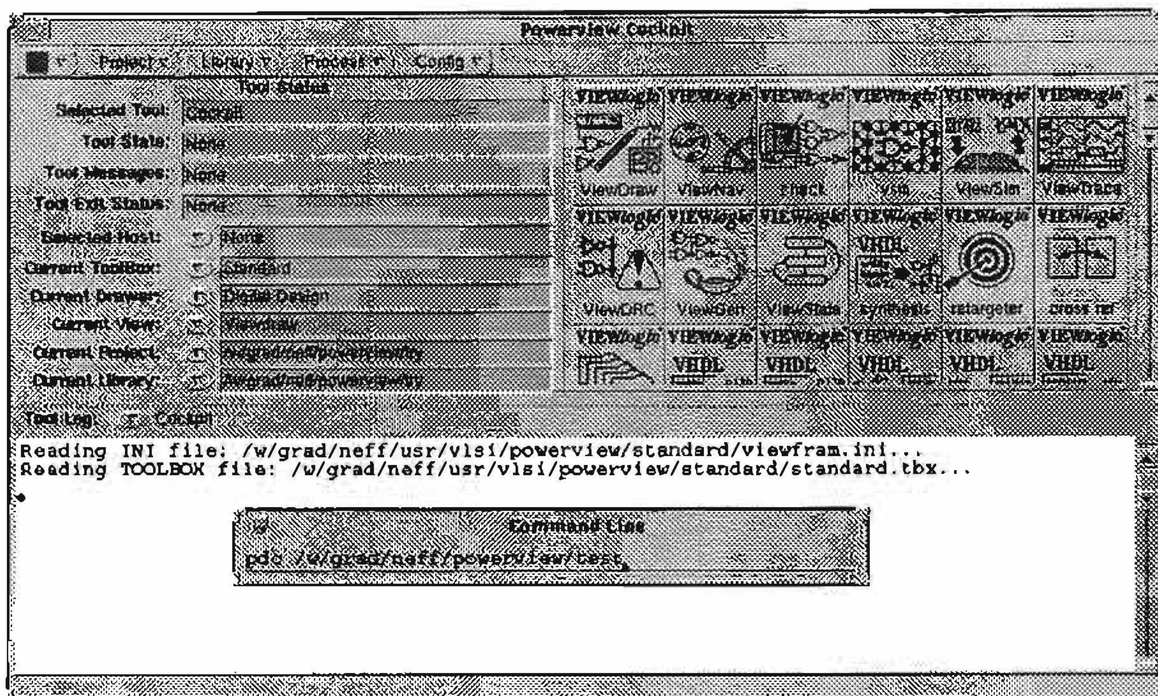


Figure 5.36. Automatic Powerview Project Creation

must be running in the background so that it can be sent commands in its interactive mode by **xse**, which is running in the foreground.

8. When **powerview** has finished this automatically-requested project creation task, because the `$(PROJDIR)/wir` subdirectory will then exist, the check-sleep-check-again loop will exit, whereupon **xse** sends the button click and confirming Return key event necessary to quit **powerview**.

Figure 5.36 shows the **powerview** window with its command line entry box exposed and stuffed with the characters (minus the final Return) sent to it by **xse**.

Ironically, of all the vistafying agents and interactions described in this chapter, the by far most useful ones (**adsim** and **dsim**) are also the ones with empty schematics, that is, those with no cells and no wires. Hence, no codified C++ vistafications are warranted, as purely textual, executable shell scripts suffice to specify the necessary interactions. As these scripts could just as easily be executed from the command line of any interactive Unix shell, it is only the desire to maintain a consistent graphical user interface that motivates the creation of a special protolib for storing these scripts. In the context of this protolib, called **exe**, when invoking the Vistafy command, only the *Run* option need be exercised, as the executable script is already there. Since it has the same base name as the empty `.cm` file, the Executer simply invokes this script with the parameters entered by the user in the Vistafy command dialog box. To justify its creation, other than just for identifying the desired script by its name, the `.cm` file could profitably be used to store and display documentation about the script. It could thereby instruct or remind the

user, who would see this documentation when loading the `.cm` file into Acme, what parameters to enter, and what they mean.

CHAPTER 6

CONCLUSION

The purpose of this chapter is two-fold; to recapitulate Vista's contributions, and to discuss future prospects for this research.

6.1 Research Contributions

In its stated purpose, versatile tool-agent interaction specification, there are three main contributions of Vista, listed in increasing order of significance:

1. A conceptual and implementational framework and protocol for the executable specification of tool-agent interaction.
2. A means to more productive use of *egocentric* tools, through a rich synthesis of encapsulated interaction mechanisms.
3. A collection of design patterns containing knowledge about how to use Vista to solve new tool interaction problems, and an algorithm for applying them.

To address the quality of versatility, it has been amply demonstrated just how widely applicable is the high-level protocol embodied in Vista's orchestrating quartet of Codifier, Analyzer, Manifester and Executer. Controlled and coordinated by special-purpose, behind-the-scenes Vistafiers, cells push and pull wires through their state space. Cells implement a variety of high-level protocols that enable the specification of many tool-agent interactions using a variety of different mechanisms. The extent of the horizontal and vertical versatility thus far attained by Vista is plotted in Figure 6.1. The legend for the vistafication data points with cross-references to where each is discussed follows:

1. sig, test (Section 4.3, page 48)
2. fsp, lswc (Section 5.1, page 79)
3. fsp, stuqwc (Section 5.1, page 79)
4. fsp, howmanydups (Section 5.1, page 79)
5. fsp, gnuplotvar (Section 5.1, page 79)
6. xse, send (Section 5.2, page 101)
7. unx, howmanyfiles (Section 5.3, page 104)
8. unx, exor (Subsection 5.3.1, page 107)

	signals	files	streams	pipes	events	sockets
acme	1,14	1-14			8	
alert				4,12,13	4,12,13	
awk		1,3,4,7,9,10,11,14		4,9		
cat		1,10,11,12				
cp		12,13				
csh		10,11	10,11	10,11		
cut				1,3,4,7,9,10,11		
dc				4		
diff		1,3,4,7,9,10,11				
echo		1,10-13	12,13	4		
emacs		1,12-14	14	14		
g++		1-11				
gdb		14			14	
gnuplot		5	5	5		
grep		1,3,4,7,9,10,11		7,11		
head		12,13				
hspice		12				
join		10,11				
ls		2,7,9	2	2,7		
madsnet		12				
madssim		12				
make		1-13				
mv		12,13				
oleo		11	11		11	
paste				9		
powerview					12,13	
rm		1,3,4,7,9,10-13				
sc		11	11		11	
sed		10-13		1,3,4,7,9,10,11		
sh		14		14		
simppl		8				
sort		3,4		1,3,4,7,9,10,11		
spplice		8,12,13				
tail		12,13				
test		1,3,4,7,9,10-13				
tr				1,3,4,7,9,10,11		
uniq		1,3,4,7,9,10,11		1,3,4,7,9,10,11		
viewsim		12,13				12
viewtrace		12,13				
wc		3	2	2,3,4,7		
wish		5,14	14			
xargs		1,3,4,7,9,10,11		9		
xedit		7			7	
xse		12,13	8			
xsekey		12,13		12,13		
xterm		11,12	11		8,11	
xwininfo			8,11			
X (server)					6	
Unix (kernel)	1					10,11

Figure 6.1. Vista Versatility Matrix

9. unx, ct2code (Subsection 5.3.2, page 109)
10. cts, sxe (Subsection 5.5.1, page 123)
11. sse, tts (Section 5.6, page 139)
12. exe, adsim (Section 5.7, page 149)
13. exe, dsim (Section 5.7, page 149)
14. exe, rootdump (Section 6.2, page 158)

Addressing the issue of productivity is harder, owing to the problematic nature of productivity metrics. In programming, for example, the "Lines-of-Code" (LOC) measure of productivity is infamous.[1] Often blurred in studies of programming productivity is the distinction between *economic* productivity and *common* productivity. The former refers to goods or services produced per unit of labor or cost. The latter simply means completing a task as quickly as possible. It is the latter meaning that is germane to the claim that more productive use of egocentric tools is made possible by Vista.

Before examining tool-use productivity, it may be helpful to use LOC as a measure of *value added* (i.e., improvements or enhancements) in the Vista context. A variation of the VAR (Value Added Reseller) acronym is the VAR of Value Added Reuse[2] (of existing software). Thus, it may also be enlightening to report the VA and the VAR in the following three phases of Vista's evolution, which phases were not distinct and sequential, but rather blurred and interwoven:

1. adding code to Acme,
2. building protolibs, prototypes, wrappers, etc., and
3. finding and testing supporting code (*var*, *coral*, etc.).

For example, the code written in the second phase is all VA, while the VAR of the *var* library consists of a mere 30 LOC added to implement the *VarMapIterator* class. To put it all in perspective, Table 6.1 tabulates the VA/VAR for phases one and three.

The small VA of 12, 19 and 6 LOC for the InterViews Dispatch library, **coral** and **Explain**, respectively, consists of some bug fixes, together with the necessary additions of `#ifdef` conditional compilation directives and extra `#include` directives to enable the g++ (GNU's C++) compiler to compile these tools and library. The larger 539 LOC VAR of the **xse** *tool* is what makes it into a useful *library* as well, by adding both the necessary interfaces to the core functions already there,

[1]See [42], which discusses the problems of measuring software, and examines the associated paradoxes, such as the fact that traditional "Lines-of-Code" (per unit of time) measures penalize high-level languages, and frequently move in the wrong direction as productivity increases.

[2]The Great Promise of OOP, which for pragmatic reasons lies largely unfulfilled, is that of *code reuse through inheritance*. This promise will perhaps be well on its way to fulfillment on the day when productivity is measured not in lines of code written, but in lines of code *that did not have to be written*, because it was already there and could be readily reused.

and the additional functionality not already there, e.g., hashing, inter-event delay specification and event list creation and traversal.

Name	Tool	Library	Original LOC	Added LOC	VA	VAR
acme	x		111220	1413	x	
IV Dispatch		x	3639	12	x	
coral	x		53588	19	x	
Explain	x		3373	6	x	
var		x	2305	30		x
fsm		x	431	12		x
xse	x	x	3624	539		x

Table 6.1. Lines of Code Value Added and Value Added Reuse

Grafting Vista onto Acme has been beneficial in two ways. First, many bugs were discovered through exercising Acme more strenuously than usual. Second, Acme's capabilities were extended and its user interface improved in the process of writing and testing Vista. For example, as befits a symbiotic relationship, the development of the Vista **xse** protolib both lent to and borrowed from Acme's incorporation of event journaling (recording and playing back X mouse and keyboard events) capabilities. Port typing was similarly mutually beneficial, enabling, for instance, the "special" typing[3] of ports and wires.

The ease with which changes to Acme were made validates the careful attention given by the Acme development team to data abstractions, object models, mechanisms and paradigms.[4] As a result, it was possible to implement the Vista framework in only 1413 lines of code, or a little over 1% of Acme's total code. Moreover, the ease with which the Acme-symbiotic part of Vista was implemented validates the fundamental Acme philosophy of simultaneously supporting different design techniques—physical, structural and behavioral. That is, just as Acme allows both physical design and structural design (schematic capture) to be performed simultaneously by the designer, so now Vista allows the simultaneous specification of structure and behavior in a versatile range of applications.

Predominantly in phase two, Vista has successfully leveraged the three mainstays of object-oriented programming—encapsulation, inheritance and polymorphism.

[3]Originally, typing ports and wires "special" was meant to facilitate mixed analog/digital design. As analog power busses are wider than ordinary signal wires, it is helpful to distinguish them by drawing the specially-typed analog wires with a wider brush. This feature is potentially useful in providing visual cues for Vista wires as well. For example, a quantitative attribute of any communication channel is its bandwidth, or data-carrying capacity. Wires representing higher-bandwidth channels could be drawn proportionally wider than lower-bandwidth wires.

[4]For example, the separation of interface and implementation in a cell prototype mirrors the separation of interface and implementation in C++ classes.

By providing a lot of small, simple components, instead of a few large, complex object conglomerates, Vista avoids the degradation of performance characteristic of lowest common denominator approaches. Vista also avoids *straining at a gnat and swallowing a camel*, or the syndrome where the complexity of the solution is all out of proportion to the complexity of the problem. Each component is small enough to be easily grasped. None of the cell or wire class definitions exceeds one page of code; indeed, the average is around 15 LOC. *UnixMIO* is the heavyweight at 49 LOC, four of which are comments. Exactly one of the sample vistafications (**sxe**) required more than 30 seconds[5] of real time to compile and link. The average was about 21 seconds, with the fastest one completing in 12 seconds.

Obviously, like anything else, Vista cannot be all things to all people, especially those interested in software design in general. Instrumentation and infrastructure have been its primary provinces; user experimentation is encouraged and facilitated by Vista, while theory and formality are basically ignored. Vista patterns (presented in the dissertation) are an incipient but admittedly not a definitive solution to the problem posed by the subjective subject of the next section.

6.2 The Right Choice

The Right Choice is the idea that if doing something is both possible and desirable, then that something is in some sense "right" and should be done. While not meant in the strong sense of a binding moral obligation, for instance, if *finding the right tool* for the job is possible (debatable) and desirable (indisputable), then it behooves one to do it. Vista subscribes wholeheartedly to this credo; in fact, it is the sustained, painstaking vigilance Vista has given to this quest that has provided the means by which Vista users can make more productive use of their time. When a vistafied tool-interaction task that previously required time-consuming manual intervention now happens automatically, this is nothing but net benefit for users.

The apotheosis of this thesis can be seen in a subtask performed by **adsim**, which uses **emacs** in batch mode to automatically edit the simulation input file.[6] In this case, **emacs** really is the right tool for the job, as to do it another way, for example, using pipelines with **sed**, or **awk**, etc., is complicated and inefficient. Vista, via **adsim**, encapsulates the small number of *elisp* commands needed to perform these batch edits, and saves the user the trouble (but does not take away the option) of learning them.

Many other tedious editing tasks have been automated by batch-mode invocation of **emacs**, such as setting the current **powerview** project, a task that to do using

[5] 31, to be exact, on an unloaded Sun SPARC 2 workstation.

[6] Of course, the user must still provide the stimulus and probe statements, but these can be put in a separate file that **emacs** inserts in the input file when making the other changes required to mollify **madssim** (and **hspice**). The user need no longer expend time and effort on every iteration (and simulation is a task that necessitates more than a few iterations) making sure the numerous nuances of the input file are correct and consistent. Considerately, however, **adsim** pauses before giving **madssim** the go ahead and affords the user an opportunity to inspect, and if necessary or desirable, to further edit the input file.

powerview requires manual menu picks, with no keyboard shortcuts to speed it up. While it could be done automatically using synthetic X events, it is so much faster to simply edit the text file where **powerview** stores a project list and an index into that list indicating which project is the current one. An older version of **adsim** used two temporary files in conjunction with **grep**, **cut**, **echo**, **dc**, **tail** and **mv** to do what **emacs** can do faster with five lines of *elisp* code.

Another example of the right choice of tool is found in the spreadsheet interaction of Section 5.6. Here, using C++ to implement the external **dollars** program is highly preferable to trying to implement the same algorithm directly in a spreadsheet program such as **sc**. While possible, it would be extremely cumbersome and slow, and thus highly undesirable, especially given the elegant C++ solution (see Appendix B) that uses recursion, which spreadsheets cannot do.

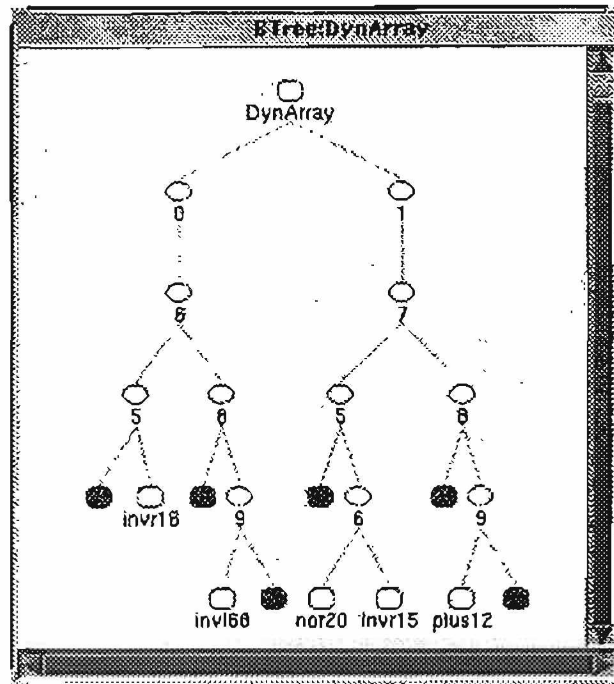
A prime example of a very useful interaction specification, **rootdump** involves the following choice of tools, and *crucial knowledge of how to use them*: **acme**, **emacs**, **gdb**, **awk**, **sh** and Tcl/Tk. The goal of this tool interaction was to visually debug some Acme code that was causing corruption of some of its "btree" data structures. Reproduced in Figure 6.2 are two results of the **rootdump** tool interaction, showing the ease with which bad trees can be distinguished from good ones, given the right tools and the right know-how. The two key steps in **rootdump** were to:

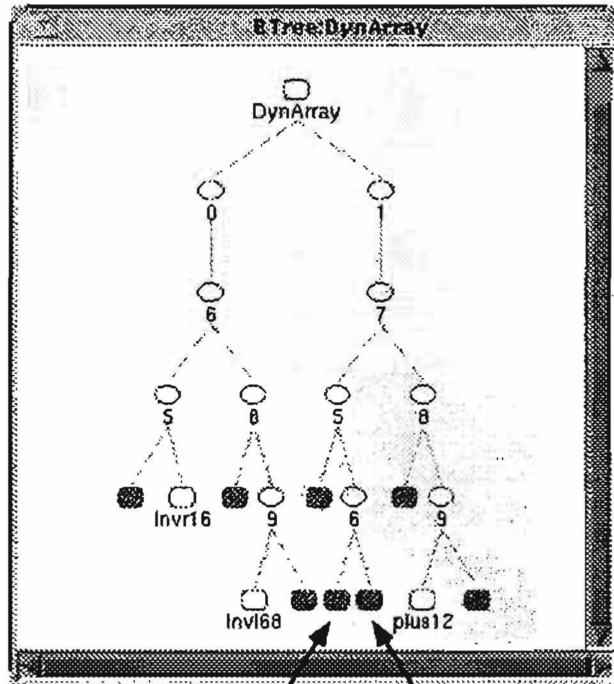
1. Take the Acme *DynArray::Dump* method that prints to standard output a textual description of the "dynamic" array of "root" (btree) pointers for the current cellmatrix, and modify this method to write its output to a file.
2. Put a **gdb** (GNU debugger) breakpoint on the *DynArray* method called whenever the current cellmatrix view is refreshed, and using its command language, define the following two operations for **gdb** to carry out when the breakpoint is reached:

   ```
   call this->Dump("rootdump", 0)
   shell mkbtree < rootdump | sh
   ```

   The first of these two commands calls the *DynArray::Dump* method, telling it to put its output in the "rootdump" text file. As **gdb** can invoke the Unix shell, the second command tells it to run the **mkbtree** program, redirecting its input from the "rootdump" file. A 44-line **gawk** (GNU's **awk**) script, **mkbtree** converts the textual "rootdump" into a list form prepended with the name "btree" and followed by an ampersand, which it puts to its standard output. Finally, this output is piped to another invocation of the shell, which runs the **btree** Tcl/Tk script in the background. The **btree** script is a modification of a public domain directory browser implemented in Tcl using Tk and a tree "widget" (implemented in C++) for displaying dynamic trees.

Yet another task where know-how is most valuable is that of making a piece of code more useful or efficient, by carefully considered modifications, that though





Null cell in both left leaf and right leaf invalidates tree.

Figure 6.2. Visual Root Dumps, Good and Bad

minor in size may be major in impact. For example, the "ordered joining" modification to **dispatch** (mentioned in Subsection 5.3.2) required the addition of just 22 lines of code, interspersed throughout the `dispatch.c` file (see Appendix B).

Still another place where small changes have dramatic repercussions is in the *Stripper* class described in Section 5.4, and used in the "Comments to Code" **ct2code** vistafication in Subsection 5.3.2. Shown in Figure 6.3 is an abridged definition of the *Stripper* class, including just the two additional member variables, the four methods modified to use them (*PutChar*, *PutSlash*, *GetChar* and the constructor), and the sole additional method, the *GetCount* accessor. Figure 6.4 contains the main function modified to use a file argument interface, while retaining the original stream input interface.

What these minor modifications do is to merge the functionality of **wc -c** into the C++ comment stripper FSM, so that instead of writing out the non-comment characters as it recognizes them, it merely counts them, since the final count is what is desired. Thus, instead of "`stripper < $file | wc -c`" it suffices to say "`stripper $file`" to get the number of strictly code characters in the specified file. Requiring 27 minutes (64%) less real time to generate with the counting stripper than with the non-counting stripper, the output of the **ct2code** vistafication is here reproduced:

```
Author    Code+Cmnts    Code Only    Cmnt2Cd

 ????       574049       415350        0.38
 Brad       550956       412934        0.33
 Mike        90805        84702        0.07
 Rick      1161921       957061        0.21
 Tony       769111       668357        0.15
```

In reality, the `~/acme/sources/*/*/*.[ch]` pathname pattern used by this code is not quite adequate, as it overlooks some of Acme's source files. For pragmatic reasons at a certain point in Acme's development, most of its `.h` files were split up, and portions put into two additional files having a different extension but the same base name as the `.h` file. No author-identifying comment was put into these files, as their author is the same as the original `.h` file's author. Making a multifurcated pipelined vistafication that accounts for this additional complexity is substantially harder and rapidly reaches the point of diminishing returns. By comparison, included in Appendix B are two alternative script implementations (using **gawk** (GNU's **awk**) and Tcl) that are almost a toss-up as to which uses the right tool to accomplish this more difficult task. As it was trivial to program, both scripts produce the following more informative output:

```
class Stripper : public FSMContext {
private:
  ifstream* ifs; int count;
public:
  Stripper(istream& i, ostream& o, char* file = nil)
    : i(i), o(o), slash('/'), star('*'), eol('\n') {
      ifs = (file == nil) ? nil : new ifstream(file);
      count = 0; EnterState("Start"); Start();
  }
  void PutChar(void) {
    if (ifs != nil) count++; else o << c;
  }
  void PutSlash(void) {
    if (ifs != nil) count++; else o << slash;
  }
  boolean GetChar(void) {
    return (ifs != nil) ?
           (ifs->get(c) != NULL) : (i.get(c) != NULL);
  }
  int GetCount(void) {
    return count;
  }
};
```

Figure 6.3. Modified Stripper Class Definition

```
int main(int argc, char** argv) {
  char* file = ((argc == 2) ? argv[1] : nil);
  Stripper stripper(cin, cout, file);
  while (stripper.GetChar()) {
    stripper.GotChar();
  }
  if (file != nil) {
    cout << stripper.GetCount() << endl;
  }
  return 0;
}
```

Figure 6.4. Modified Stripper Main Function

	Total Files	Total Lines	Total Bytes	% Comments	% Code	Ratio
????:	82	19215	586995	0.27	0.73	0.38
Brad:	103	19927	626972	0.23	0.77	0.31
Mike:	23	3717	96344	0.07	0.93	0.08
Rick:	140	40968	1206197	0.17	0.83	0.21
Tony:	88	28806	825553	0.13	0.87	0.15
	436	112633	3342061			

Finally, because the full power and flexibility of the Unix shell is at its disposal, as are all the varied tools and vistafied agents that must be made to interact, **make** is the right choice of tool for a great many tasks. In recognition of their significant contribution, all the makefiles used directly or indirectly by Vista (including `adsim.mk` and `dsim.mk`) are showcased in Appendix C. To show how Vista can be made to aid **make**, two readily written special-purpose wire and cell classes, *Dependency* and *Target*, are presented. Their purpose is to generate a makefile template from a visual depiction of all target-dependency relationships. These two class definitions are shown in Figure 6.5, and, for an example of their utility, the "dsim" schematic and the text generated by the execution of its vistafication appear in Figure 6.6 and Figure 6.7 respectively.

In sum, still to a significant degree a subjective matter of intuition and inspiration, still more an art than a science, choosing

- the Right Tools,
- the Right Techniques,
- the Right Abstraction Levels, and
- the Right Behavior Partitioning

is as difficult as it is rewarding. Though impossible to pinpoint precisely, the magnitude of the effort and the munificence of the reward fall somewhere between that of choosing the Right Word and choosing the Right Mate.

6.3 The Constancy of Change

The relentless pressure of change applies to the little picture of cells mutating wires, as well as to the big picture of future prospects for Vista, or indeed, any software project. Truly, the pervasiveness of change is older than time, and larger than space. Nowhere nowadays is change more evident than in software—its architectures, environments and methodologies. From every side, changing paradigms, protocols and principles continue to bombard programmers and analysts. Keeping abreast of the latest developments is like surfing on a tidal wave. Software users and information consumers will be insulated from the chaos of change only to the extent that these programming practitioners succeed in creating the right kind of interfaces to the right kind of computing services.

No less a guru than Alan Kay has predicted that user interfaces of the 1990s will not be *tool*-based as in the 1980s, but will instead be *agent*-based. Envisioned is an army of autonomous agents, possessing varying degrees of (artificial)

```
class Dependency : public Wire {
protected:
  VarMap p, c;
  boolean this_is_connected;
public:
  Dependency(const char* name)
    : Wire("Dependency", name), this_is_connected(true) {
  }
  Dependency(void) : Wire(), this_is_connected(false) {
  }
  var GetAllKeys(VarMap& vm, var separator) {
    var result;  VarMapIterator vmi;
    vmi(&vm);
    while (vmi++) {
      result += separator;
      result += vmi.key;
    }
    return result;
  }
  char* Parents(void)  { return (char*) GetAllKeys(p, " "); }
  char* Children(void) { return (char*) GetAllKeys(c, " "); }

  void operator >> (const char* name) {
    if (this_is_connected) {
      p.append(name, ""); // name identifies a parent
    }
  }
  void operator << (const char* name) {
    if (this_is_connected) {
      c.append(name, ""); // name identifies a child
    }
  }
} vvparents, vvkids, vvchildren, vvdependents;

defCell(Target, (NAME, BIDIR(Dependency, vvchildren),
                       BIDIR(Dependency, vvdependents),
                       BIDIR(Dependency, vvkids),
                       BIDIR(Dependency, vvparents)))
  if (VISTAFIER.IsAnalyzing()) {
    vvchildren   >> name;
    vvdependents >> name;
    vvkids       >> name;
    vvparents    << name;
  } else if (VISTAFIER.IsExecuting()) {
    cout << name << ':' << vvparents.Parents() << endl;
    cout << '\t' << "@echo Making " << name << endl;
    cout << '\t' << "@touch " << name << endl << endl;
  }
```

Figure 6.5. *Dependency* and *Target* Class Definitions

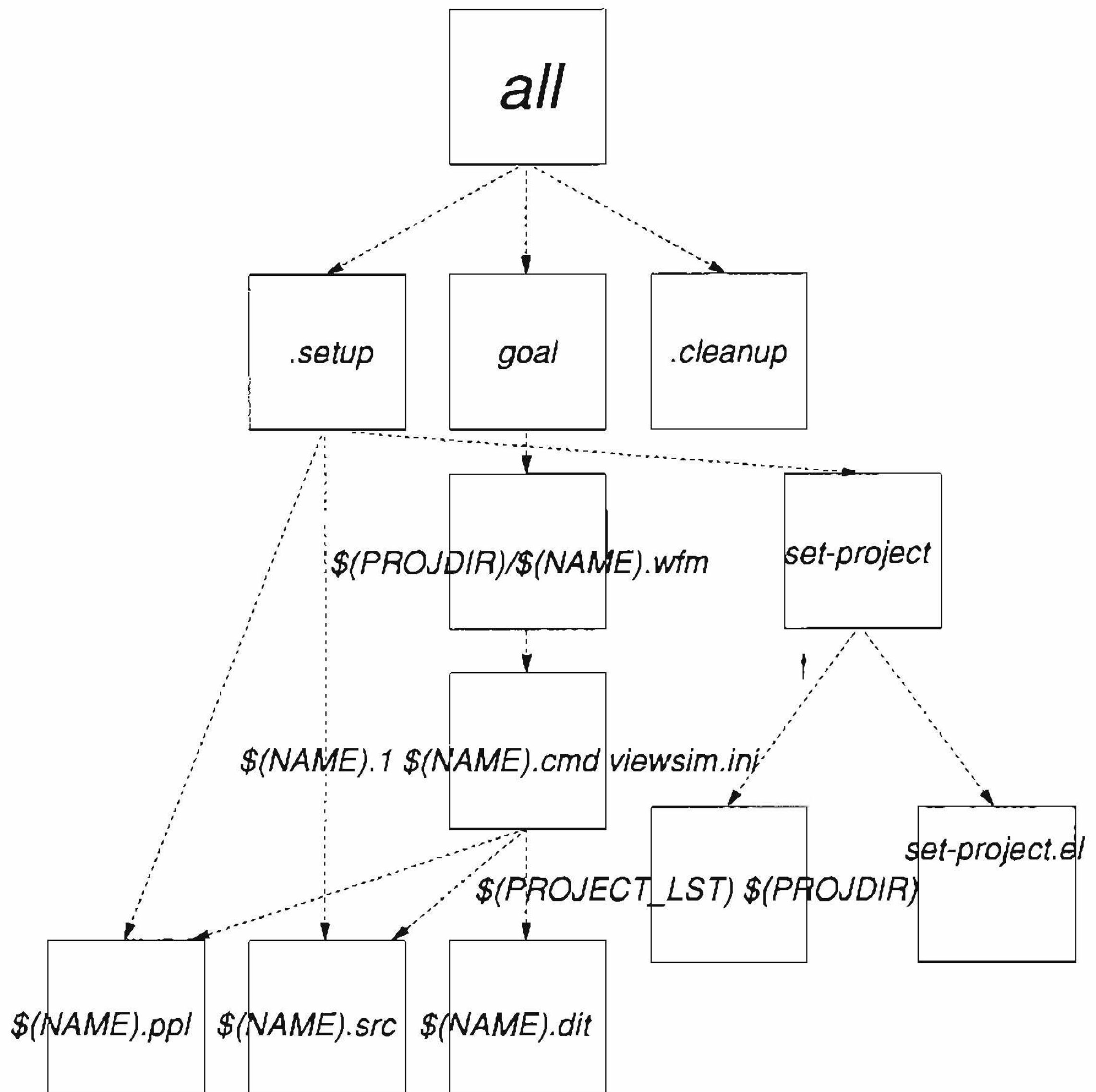


Figure 6.6. Targets and Dependencies for "dsim"

```
all: .setup goal .cleanup
	@echo Making all
	@touch all

.setup: set-project set-project $(NAME).ppl $(NAME).src
	@echo Making .setup
	@touch .setup

goal: $(PROJDIR)/$(NAME).wfm
	@echo Making goal
	@touch goal

.cleanup:
	@echo Making .cleanup
	@touch .cleanup

$(PROJDIR)/$(NAME).wfm: $(NAME).1 $(NAME).cmd viewsim.ini
	@echo Making $(PROJDIR)/$(NAME).wfm
	@touch $(PROJDIR)/$(NAME).wfm

set-project: $(PROJECT_LST) $(PROJDIR) set-project.el
	@echo Making set-project
	@touch set-project

$(NAME).1 $(NAME).cmd viewsim.ini: $(NAME).ppl $(NAME).src $(NAME).dit
	@echo Making $(NAME).1 $(NAME).cmd viewsim.ini
	@touch $(NAME).1 $(NAME).cmd viewsim.ini

$(PROJECT_LST) $(PROJDIR):
	@echo Making $(PROJECT_LST) $(PROJDIR)
	@touch $(PROJECT_LST) $(PROJDIR)

set-project.el:
	@echo Making set-project.el
	@touch set-project.el

$(NAME).ppl:
	@echo Making $(NAME).ppl
	@touch $(NAME).ppl

$(NAME).src:
	@echo Making $(NAME).src
	@touch $(NAME).src

$(NAME).dit:
	@echo Making $(NAME).dit
	@touch $(NAME).dit
```

Figure 6.7. Generated Makefile Skeleton for “dsim.mk”

intelligence, empowered and commissioned to perform the drudgery—the tedious and time-consuming tasks that humans now find necessary to do themselves. For example, net-mining agents could scan the vast repositories of information available in a networked computing environment such as the Internet, filter out the data of little or no interest to their human masters, format and offer to them the subset deemed valuable for inspection at their leisure. Such agents will presumably need to communicate with a number of general- and special-purpose tools in order to carry out their tasks. A clue to how such tools and agents may interact can be found in a report on an international workshop on programming environments, where appeared the following abstract entitled *Tool Integration Technologies Through the 90's* [38], which, though singling out CASE frameworks, forecasts a widely applicable technique:

> The problem of integrating separately written tools so that they work together cooperatively is recognized as a key issue in CASE frameworks. The existing model for building tools emphasizes separate large components, sharing rigid models of the data they manipulate. In order to construct tools which can be more readily mixed and which can meet the demand for more well-integrated visual environments, we see trends toward a finer grain-size for both data elements and control elements and from a procedural control-flow approach toward a compositional, object-oriented tool construction.

The rising generation of tools has great opportunities for rallying to this cooperative cause. The current generation of egocentric tools need not be written off quite yet, however, insofar as interim solutions like Vista can facilitate and meliorate their interaction. With a major emphasis being placed on the preservation of so-called *legacy* software, which constitutes an enormous investment for many companies, the need for aids like Vista, which *must be* small and flexible enough to adapt quickly to changing environments and requirements, will likely continue for the duration of the legacy.

APPENDIX A

VISTA CLASS LIBRARIES

```
// File: vista.h

#include <stream.h>
#include <fstream.h>
#include <stdlib.h>
#include <unistd.h>
#include <var/var.h>
#include <var/varmap.h>

#if !defined(boolean_defined) && !defined(boolean)
#define boolean_defined
typedef unsigned boolean;
#define false 0
#define true  1
#endif

#ifndef nil
#define nil 0
#endif

#define self (*this)

#define defCell(n, args) \
class n : public Cell {\
public:\
  n args : Cell(#n, name) {

#define outclude }};

#define MAIN_BEGIN \
int main(int argc, char** argv) {\
  VISTAFIER.Init(argc, argv);

#define MAIN_END \
  return VISTAFIER.Exit();\
}

#define MAIN_BEGIN_CELLS \
  do {
```

```
#define MAIN_END_CELLS \
  } while (VISTAFIER.IsVistafying());

#define NAM const var
#define NUM (double)
#define STR (char *)
#define NAME const char* name
#define MODS var mods
#define BIDIR BIPUT
#define BIPUT OUTPUT
#define OUTPUT(typ, nam) typ& nam
#define INPUT(typ, nam) typ& nam

#define BPORT(cellname, portname) portname.Port('b', cellname, #portname)
#define IPORT(cellname, portname) portname.Port('i', cellname, #portname)
#define OPORT(cellname, portname) portname.Port('o', cellname, #portname)

ofstream vistalog; // global for access by Vistafier, Cell and Wire classes
```

```
////////////////////////////////////////////////////////////////////////////////
//
// Vistafier is the base class for all "protolib" classes.  Each protolib
// has a .h file, near the beginning of which should appear the following:
//
// #undef VISTAFIER
// #define VISTAFIER <protolib_name>
//
////////////////////////////////////////////////////////////////////////////////

class Vistafier {
protected:
  enum { ANALYZING, MANIFESTING, EXECUTING, DONE } state;
  boolean show;
  var basename, dirpath;
  int status;
public:
  Vistafier(void) : state(ANALYZING), show(true), status(0) {
  }
  void Init(int argc, char** argv) {
    char* name = argv[0];
    char* lastslash = strrchr(name, '/');
    if (lastslash != nil) {
      name = lastslash + 1;
    }
    basename = name;
    if (argc >= 5) {
      show = (strcmp(argv[4], "1") == 0);
    }
    if (argc >= 6) {
      const char* path = (const char*) argv[5];
      if (chdir(path) == -1) {
        cerr << "Cannot chdir to " << path << " --- no such path." << endl;
        exit(1);
      }
      dirpath = path;
    }
    char* logfilename = getenv("VISTALOG");
    if (logfilename != nil && logfilename[0] != '\0') {
      vistalog.open(logfilename);
    }
    vistalog << "Vistafier(\"";
    if (argv == nil) {
      vistalog << argc;
    } else {
      for (int i = 1; i < argc; i++) {
        if (i > 1) vistalog << ' ';
        vistalog << argv[i];
      }
    }
    vistalog << "\")" << endl;
  }
```

```
  int Status(void) {
    return status;
  }
  virtual void MakeManifest(void) {
    if (show) {
      var makecmd("make -r -C ");
      makecmd += dirpath;
      makecmd += " manifest OUT=";
      makecmd += basename;
      status = system(STR makecmd);
      if (status > 0) state = DONE;
    }
  }
  virtual void NextState(void) {
    if      (state == ANALYZING)   state = MANIFESTING;
    else if (state == MANIFESTING) state = EXECUTING;
    else if (state == EXECUTING)   state = DONE;
  }
  virtual boolean IsAnalyzing(void) {
    return (state == ANALYZING);
  }
  virtual boolean IsManifesting(void) {
    return (state == MANIFESTING);
  }
  virtual boolean IsExecuting(void) {
    return (state == EXECUTING);
  }
  virtual boolean IsDone(void) {
    return (state == DONE);
  }
  virtual boolean IsVistafying(void) {
    if (IsAnalyzing()) {
      vistalog << "========" << endl; // indicates end of analyzing output
    }
    if (IsManifesting()) {
      MakeManifest();
      vistalog << "--------" << endl; // indicates end of manifesting output
    }
    NextState();
    return (state == MANIFESTING || state == EXECUTING);
  }
  virtual int Exit(void) {
    return status;
  }
  virtual ~Vistafier(void) {
  }
};
```

```
class Cell {
protected:
  var t, n; // type and name of cell
public:
  Cell(const char* type, const char* name) : t(type), n(name) {
    vistalog << type << '(' << '"' << name << '"' << ')' << endl;
  }
  const char* GetType(void) {
    return (const char*) t;
  }
  const char* GetName(void) {
    return (const char*) n;
  }
};

class Wire {
protected:
  var t; // type of wire
  var n; // name of wire
  var v; // value of wire (sort of)
public:
  void Init(const char* type, const char* name, int value) {
    t = type;
    n = name;
    v = value;
    vistalog << type << '(' << '"' << name << '"' << ')' << endl;
  }
  void Init(const char* type, const char* name, const char* value) {
    t = type;
    n = name;
    v = value;
    vistalog << type << '(' << '"' << name << '"' << ')' << endl;
  }
  Wire(const char* type, const char* name) : t(type), n(name), v(name) {
    vistalog << type << '(' << '"' << name << '"' << ')' << endl;
  }
  Wire(void) {
  }
  virtual ~Wire(void) {
  }
  const char* GetType(void) {
    return (const char*) t;
  }
  const char* GetName(void) {
    return (const char*) n;
  }
  void Port(char mode, const char* cellname, const char* portname) {
    vistalog << mode << "port(\"" << cellname << '_' << portname
             << "\",\"" << cellname << "\")" << endl;
    vistalog << "terminal(\"" << cellname << '_' << portname
             << "\",\"" << GetName() << "\")" << endl;
  }
```

```
  boolean operator ! (void) {
    return v.is_string();
  }
  Wire& operator = (int i) {
    v = i;
    return self;
  }
  Wire& operator += (int i) {
    if (v.is_string()) {
      v = i;
      v.change_type("int");
    } else {
      v += i;
    }
    return self;
  }
  Wire& operator += (const char* s) {
    v += s;
    return self;
  }
  double operator * (void) {
    return (double) v;
  }
  const char* operator () (void) {
    return (const char*) v;
  }
  char& operator [] (int i) {
    unsigned int l = v.length();
    if (i >= 0) {
      return v[i%l];
    } else {
      return v[l-((-i-1)%l)-1];
    }
  }
};

ostream& operator << (ostream& os, Wire& wire) {
  if (!wire) {
    os << wire();
  } else {
    os << *wire;
  }
  return os;
}
```

```
class VarMapIterator {
private:
  VarMap* vm;
  boolean at_end, did_end;
public:
  var key;
  var val;
  VarMapIterator(void) {
    key.format("%s");
    val.format("%s");
  }
  void operator () (VarMap* varmap) {
    vm = varmap;
    vm->first();
    at_end = did_end = vm->empty();
  }
  boolean operator ++ (void) {
    if (at_end && did_end) {
      return false;
    } else if (at_end) {
      did_end = true;
    } else if (vm->at_end()) {
      at_end = true;
    }
    key = vm->key();
    val = vm->value();
    vm->next();
    return !did_end;
  }
};
```

```
// File: sig.h

#include "vista.h"
#include <signal.h> // for signal() and kill()
#include <unistd.h> // for pause()

class Medium : public Wire {
private:
  int signaler_pid, signalee_pid;
public:
  Medium(const char* name) : Wire("Medium", name) {
    signaler_pid = signalee_pid = 0;
  }
  ~Medium(void) {
  }
  Medium& operator << (int pid) {
    if (signaler_pid == 0) {
      signaler_pid = pid;
    } else {
      signalee_pid = pid;
    }
    return self;
  }
  Medium& operator >> (int& pid) {
    if (signalee_pid != 0) {
      pid = signalee_pid;
      signalee_pid = 0;
    } else {
      pid = signaler_pid;
      signaler_pid = 0;
    }
    return self;
  }
};

#undef VISTAFIER
#define VISTAFIER sig

class VISTAFIER : public Vistafier {
private:
  var message;
  int signaler_pid, signalee_pid;
public:
  void Init(int argc, char** argv) {
    Vistafier::Init(argc, argv);
  }
  VISTAFIER(void) : signaler_pid(0), signalee_pid(0) {
  }
  ~VISTAFIER(void) {
  }
```

```
static void GotSignalAlarm(void) {
  // dummy function used only for "unpausing"
  // on receipt of SIGALRM signal
}
int GetSignalerPid(var id) {
  if (id == "self") {
    return getpid();
  } else {
    return 0;
  }
}
int GetSignaleePid(var id) {
  if (id == "parent") {
    return getppid();
  } else {
    // should query kernel via system("ps | awk")
    // to find pid from id
    return 0;
  }
}
int& SignalerPid(void) {
  return signaler_pid;
}
int& SignaleePid(void) {
  return signalee_pid;
}
void GetMessage(var mods) {
  int s = mods.strchr('\''), e = mods.strrchr('\'');
  message = mods(s+1, e-s-1);
}
void GetMedium(var mods) {
  int s = mods.strchr('=');
  var canrespond = mods(s+1, 1);
  if (canrespond == "1") {
    signal(SIGALRM, VISTAFIER::GotSignalAlarm); // catch alarm signals
  } else {
    signaler_pid = 0; // => signalee cannot respond
  }
}
boolean IsMessageReady(void) {
  return (message.length() > 0);
}
boolean IsMediumReady(void) {
  return (signaler_pid > 0);
}
void Send(int sig) {
  kill(signalee_pid, sig); // deliver sig to signalee process
  if (signaler_pid > 0) {  // then signalee can respond
    pause(); // wait for acknowledgement (SIGALRM)
  }
}
```

```
  VISTAFIER& operator << (int bit) {
    if (bit == 0) {
      Send(SIGUSR1);
    } else if (bit == 1) {
      Send(SIGUSR2);
    }
    return self;
  }
  VISTAFIER& operator << (char c) {
    for (int i = 0; i < 8; i++) {
      // select bit i of c
      int bit = (int(c) & (1 << i)) ? 1 : 0;
      self << bit;
    }
    return self;
  }
  VISTAFIER& operator << (char* s) {
    while (*s)
      self << *s++;
    return self;
  }
  int Exit(void) {
    if (IsMessageReady() && IsMediumReady()) {
      self << message; // deliver message
      return 0;
    } else {
      return 1;
    }
  }
} VISTAFIER;
```

```
// File: fsp.h

#include "vista.h"
#include <iostream.h>
#include <fstream.h>
#include <procbuf.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/param.h>   // which among other things does
//#define NOFILE    256  /* max open files per process */
//#define PIPE_BUF 4096  /* pipe buffer size */
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <time.h> // for usleep()
#include <sys/wait.h>

#define lowbyte(w) ((w) & 0377)
#define highbyte(w) lowbyte((w) >> 8)

#define MAXSIG 31

void statusprt(int pid, int status) {
  static char* sigmsg[] = {
    "",
    "Hangup",
    "Interrupt",
    "Quit",
    "Illegal instruction",
    "Trace trap",
    "Abort",
    "Emulator trap",
    "Arithmetic exception",
    "Kill",
    "Bus error",
    "Segmentation violation",
    "Bad argument to system call",
    "Write on a pipe with no one to read it",
    "Alarm clock",
    "Software termination signal",
    "Urgent condition present on socket",
    "Stop",
    "Stop signal generated from keyboard",
    "Continue after stop",
    "Child status has changed",
    "Background read attempted from control terminal",
    "Background write attempted to control terminal",
    "I/O is possible on a descriptor",
    "Cpu time limit exceeded",
    "File size limit exceeded",
    "Virtual time alarm",
```

```
    "Profiling timer alarm",
    "Window changed",
    "Resource lost",
    "User-defined signal 1",
    "User-defined signal 2"
  };
  int code;
  if (status != 0 && pid != 0)
    cerr << pid << " ";
  if (lowbyte(status) == 0) {
    if ((code = highbyte(status)) != 0)
      cerr << "=> " << code << endl;
  } else {
    if ((code = status & 0177) <= MAXSIG)
      cerr << sigmsg[code];
    else
      cerr << "Signal #" << code;
    if ((status & 0200) == 0200)
      cerr << "-- core dumped";
    cerr << endl;
  }
}

void wait_for_children(void) {
  int pid, status;
  while ((pid = wait(&status)) != -1) {
    statusprt(pid, status);
  }
}

void syserr(char* msg, boolean fatal=true) {
  fprintf(stderr, "ERROR: %s (%d", msg, errno);
  if(errno > 0 && errno < sys_nerr) {
    fprintf(stderr, "; %s)\n", sys_errlist[errno]);
  } else {
    fprintf(stderr, ")\n");
  }
  if (fatal) {
    _exit(1);
  }
}

void fatal(char* msg) {
  fprintf(stderr, "ERROR: %s\n", msg);
  _exit(1);
}
```

```
enum pipe_end { INPUT, OUTPUT };

class PipeEnds {
private:
  int p[2];
public:
  PipeEnds(void) {
    if (pipe(p)) {
      cerr << "PipeEnds::PipeEnds  Failed to open pipe." << endl;
      _exit(1);
    }
  }
  int operator [] (pipe_end which) {
    return p[int(which)];
  }
  virtual ~PipeEnds(void) {
  }
};

class Pipe {
private:
  PipeEnds ends;
public:
  ifstream ifs; ofstream ofs;
  Pipe(void) : ends(), ifs(ends[INPUT]), ofs(ends[OUTPUT]) {
    if (!ifs || !ofs) {
      cerr << "Pipe::Pipe  Failed to attach pipe to a stream." << endl;
      _exit(1);
    }
  }
  int operator [] (pipe_end which) {
    return ends[which];
  }
  void CloseInput(void) {
    ifs.close();
  }
  void CloseOutput(void) {
    ofs.close();
    if (ofs.fail()) syserr("close");
  }
  Pipe& operator << (unsigned char c) {
    ofs << c;
    return *this;
  }
  void SetUpForNonBlockingRead(void) {
    int fflags;
    if ((fflags = fcntl(self[INPUT], F_GETFL, 0)) == -1) syserr("fcntl");
    if (fcntl(self[INPUT], F_SETFL, fflags | O_NDELAY) == -1) syserr("fcntl");
  }
  ~Pipe(void) {
  }
};
```

```
class PipeCommand : public Pipe {
private:
  char* command;
  int len;
  int child;
  void DoPipe(void) {
    FILE* out_stream;
    if ((out_stream = popen(command, "w")) == nil) {
      cerr << "PipeCommand::DoPipe unable to issue command: "
           << command << endl;
      _exit(1);
    }
    // send everything from the input pipe to outstream
    while (!ifs.eof()) {
      unsigned char buf;
      ifs.read(&buf, 1);
      if (ifs.eof() || ifs.fail())
        break;
      fwrite((void*)&buf, 1, 1, out_stream);
    }
    fflush(out_stream);
    pclose(out_stream);
    delete [] command;
    len = 0;
  }
public:
  void SetCommand(char* com) { // not normally called by user
    if (len != 0) {
      delete [] command;
    }
    len = strlen(com);
    command = new char[len];
    strcpy(command, com);
  }
  void DoIt(void) { // not normally called by user
    // split off the processes
    switch (child = fork()) {
      case -1: {
        cerr << "fork failed" << endl;
        _exit(1);
      }
      case  0: { // child process
        // first close the output stream
        CloseOutput();
        DoPipe();
        CloseInput();
        _exit(0);
      }
    }
    // only the parent does the following
    CloseInput();
  }
```

```
  PipeCommand(char* com = nil) : len(0) {
    if (com != nil) {
      SetCommand(com);
      DoIt();
    }
  }
  ~PipeCommand(void) {
    ofs << flush;
    CloseOutput();
    wait(&child); // wait for child to exit
  }
};

#define IPIPEBUFSIZE 128

class ipipe : public Wire {
private:
  procbuf pb;
  istream p;
  unsigned char c;
  char buf[IPIPEBUFSIZE];
public:
  ipipe(const char* command)
    : Wire("ipipe", command), pb(command, ios::in), p(&pb) {
  }
  ipipe& operator >> (unsigned char& c) {
    p >> c; // delegate
    return self;
  }
  ipipe& operator >> (char* buf) {
    p >> buf; // delegate
    return self;
  }
  boolean Get(void) {
    return (p.get(c) && !p.eof());
  }
  unsigned char Got(void) {
    return c;
  }
  boolean GetWord(void) {
    return ((p >> buf) && !p.eof());
  }
  char* GotWord(void) {
    return buf;
  }
  boolean GetLine(void) {
    return (p.getline(buf, IPIPEBUFSIZE-1) && !p.eof());
  }
  char* GotLine(void) {
    return buf;
  }
};
```

```
class opipe : public Wire {
private:
  procbuf pb;
  ostream p;
public:
  opipe(const char* command)
    : Wire("opipe", command), pb(command, ios::out), p(&pb) {
  }
  opipe& operator << (unsigned char c) {
    p << c; // delegate
    return self;
  }
  opipe& operator << (const char* s) {
    p << s; // delegate
    return self;
  }
};

class pipe : public Wire {
private:
  procbuf ipb, opb;
  istream* is; ostream* os;
  unsigned char c;
public:
  pipe(const char* name) : Wire("pipe", name), is(nil), os(nil) {
  }
  boolean Get(void) {
    return (is != nil && is->get(c) && !is->eof());
  }
  void Got(void) { *os << c;
  }
  void Flow(void) { while (Get()) Got();
  }
  pipe& operator << (const char* command) {
    if (ipb.open(command, ios::in)) {
      is = new istream(&ipb);
      if (os != nil) {
        Flow();
      }
    }
    return self;
  }
  pipe& operator >> (const char* command) {
    if (opb.open(command, ios::out)) {
      os = new ostream(&opb);
      if (is != nil) {
        Flow();
      }
    }
    return self;
  }
```

```
  ~pipe(void) {
    delete is;
    delete os;
  }
};

static const int NO_FILES = 64; // getdtablesize();

class StdIO : public Wire {
private:
  int i, curi, toti, curo, toto;;
  boolean is_std;
  int ifds[NO_FILES];
  const char* srcfiles[NO_FILES];
  int ofds[NO_FILES];
  const char* dstfiles[NO_FILES];
  boolean appends[NO_FILES];
  StdIO *me, *my_shadow;
public:
  StdIO(const char* name, boolean is_standard=false)
    : Wire("StdIO", name),
  i(0), curi(-1), toti(0), curo(-1), toto(0), is_std(is_standard), me(this),
  my_shadow(nil) {
    for (int n = 0; n < NO_FILES; n++) {
      ifds[n] = 0;
      ofds[n] = 1;
      srcfiles[n] = dstfiles[n] = nil;
      appends[n] = false;
    }
    if (is_standard) {
      toti = toto = 1;
    }
  }
  StdIO* Me(void) {
    return me;
  }
  StdIO* MyShadow(void) {
    if (my_shadow == nil) my_shadow = new StdIO("my_shadow");
    return my_shadow;
  }
  int FanInCount(void) {
    return toti;
  }
  int FanOutCount(void) {
    return toto;
  }
  void IncrementFanInCount(void) {
    if (!is_std) toti++;
  }
  void IncrementFanOutCount(void) {
    if (!is_std) toto++;
  }
```

```
boolean Forks(void) {
  return (toto > 1);
}
boolean Joins(void) {
  return (toti > 1);
}
int* Ifds(boolean std = false, boolean use_me = false) {
  if (use_me || my_shadow == nil) return &ifds[std ? curi : 0];
  else return &my_shadow->ifds[std ? my_shadow->curi : 0];
}
const char** Srcfiles(boolean std = false, boolean use_me = false) {
  if (use_me || my_shadow == nil) return &srcfiles[std ? curi : 0];
  else return &my_shadow->srcfiles[std ? my_shadow->curi : 0];
}
int* Ofds(boolean std = false) {
  return &ofds[std ? curo : 0];
}
const char** Dstfiles(boolean std = false) {
  return &dstfiles[std ? curo : 0];
}
boolean* Appends(boolean std = false) {
  return &appends[std ? curo : 0];
}
void CreatePipe(int index) {
  if (ifds[index] == 0 && ofds[index] == 1) {
    // then the pipe has not yet been created for the given index
    PipeEnds ends;
    ifds[index] = ends[INPUT];
    ofds[index] = ends[OUTPUT];
  }
}
void NextIn(const char* srcfile = nil) {
  if (!is_std && !Forks() && (i < FanInCount())) {
    CreatePipe(i);
    srcfiles[i] = srcfile;
    if (Joins()) i++;
  }
}
void NextOut(const char* dstfile = nil, boolean append = false) {
  if (!is_std && !Joins() && (i < FanOutCount())) {
    CreatePipe(i);
    dstfiles[i] = dstfile;
    appends[i] = append;
    if (Forks()) i++;
  }
}
void NextI(void) {
  if (!is_std) curi++; else curi = 0;
}
void NextO(void) {
  if (!is_std) curo++; else curo = 0;
}
```

```
    int Ifd(void) {
      return ifds[curi];
    }
    int Ofd(void) {
      return ofds[curo];
    }
    const char* Srcfile(void) {
      return srcfiles[curi];
    }
    const char* Dstfile(void) {
      return dstfiles[curo];
    }
    boolean Append(void) {
      return appends[curo];
    }
    ~StdIO(void) {
      delete my_shadow;
    }
  } vvin("stdin", true), vvout("stdout", true),
    vvl("left", true), vvb("bottom", true), vvr("right", true), vvt("top", true);
```

```
#define MAXARGS 500

class UnixCommand {
private:
  boolean wait_for_me;
  int pid;
  int argc;
  char* argv[MAXARGS];
public:
  void Parse(const char* args) {
    for (int i = 0; i < MAXARGS; i++) argv[i] = nil;
    char *args_ = strdup(args);
    char *first = strtok(args_, " \t\n"), *next = nil;
    argv[argc] = strdup(first);
    while ((next = strtok(nil, " \t\n")) != nil && ++argc < MAXARGS) {
      argv[argc] = strdup(next);
    }
    free(args_);
  }
  void RedirectI(int srcfd, const char* srcfile, boolean std) {
    if (srcfd == 0 && !wait_for_me) {
      srcfile = "/dev/null";
      srcfd = -1; // so the next if will be true and /dev/null will be opened
      std = true;
    }
    if (srcfd != 0) {
      if (std) {
        if (close(0) == -1) syserr("close");
      } else if (srcfile != nil) {
        if ((srcfd = dup2(0, srcfd)) == -1) syserr("dup2");
        else close(srcfd);
      }
      if (srcfile == nil) {
        if (std) {
          if (dup(srcfd) != 0) fatal("dup");
        }
      } else if ((srcfd = open(srcfile, O_RDONLY, 0)) == -1) {
        cerr << "Cannot open " << srcfile << " for reading" << endl;
        cerr << "Using /dev/null instead" << endl;
        if (open("/dev/null", O_RDONLY, 0) == -1)
          _exit(1); // highly unlikely
      }
    }
  }
```

```
void RedirectO(int dstfd, const char* dstfile, boolean append, boolean std) {
  if (dstfd != 1) {
    if (std) {
      if (close(1) == -1) syserr("close");
    } else if (dstfile != nil) {
      if ((dstfd = dup2(1, dstfd)) == -1) syserr("dup2");
      else close(dstfd);
    }
    if (dstfile == nil) {
      if (std) {
        if (dup(dstfd) != 1) fatal("dup");
      }
    } else {
      int flags = O_WRONLY | O_CREAT;
      flags |= (append ? O_APPEND : O_TRUNC);
      if ((dstfd = open(dstfile, flags, 0666)) == -1) {
        cerr << "Cannot create/open " << dstfile << " for writing" << endl;
        cerr << "Using /dev/null instead" << endl;
        if (open("/dev/null", flags, 0666) == -1)
          _exit(1); // extremely unlikely
      }
    }
  }
}
boolean fd_in_fds(int fd, int* fds, int n) {
  for (int i = 0; i < n; i++) {
    if (fd == fds[i]) {
      return true;
    }
  }
  return false;
}
```

```
UnixCommand(const char* args,
            int nsrc, int* srcfds, const char** srcfiles,
            int ndst, int* dstfds, const char** dstfiles, boolean* appends,
            boolean wait_for_me)
  : wait_for_me(wait_for_me), pid(0), argc(0) {
  Parse(args);
  switch (pid = fork()) {
    case -1: {
      cerr << "Cannot create new process " << argv[0] << endl;
      _exit(1);
    }
    case 0: {
      // child process
      int i; boolean std = (nsrc == 1 && ndst == 1);
      for (i=0; i < nsrc; i++) {
         RedirectI(srcfds[i], srcfiles[i], std);
      }
      for (i=0; i < ndst; i++) {
         RedirectO(dstfds[i], dstfiles[i], appends[i], std);
      }
      for (int fd = 3; fd < NO_FILES; fd++) {
        if ((nsrc > 1 && (fd_in_fds(fd, srcfds, nsrc) || fd == dstfds[0])) ||
            (ndst > 1 && (fd_in_fds(fd, dstfds, ndst) || fd == srcfds[0]))) {
          continue; // do not close
        } else if (close(fd) != -1) {
          continue; // closed
        }
      }
      execvp(argv[0], argv);
    }
```

```
      default: {
        // parent process
        if (nsrc == 1 && ndst == 1) {
          if (srcfds[0] > 0) {
            if (close(srcfds[0]) == -1)
              syserr("close src");
          }
          if (dstfds[0] > 1) {
            if (close(dstfds[0]) == -1)
              syserr("close dst");
          }
        } else {
          // there is an asymmetry here---the parent dispatch join process
          // *must* close the dstfd, but the parent dispatch fork process
          // must *not* close the srcfd
          if (nsrc > 1) {
            for (int i = 0; i < nsrc; i++) {
              if (close(srcfds[i]) == -1)
                syserr("close src");
            }
            if (dstfds[0] > 1) {
              if (close(dstfds[0]) == -1)
                syserr("close dst");
            }
          }
          if (ndst > 1) {
            for (int i = 0; i < ndst; i++) {
              if (close(dstfds[i]) == -1)
                syserr("close dst");
            }
          }
        }
      }
    }
  }
  ~UnixCommand(void) {
    clog << pid;
    for (int i = 0; i <= argc; i++) {
      if (i <= 8) clog << " " << argv[i];
      if (i == 8) clog << " ...";
      free(argv[i]);
    }
    clog << endl;
    if (wait_for_me) {
      wait_for_children();
    }
  }
};
```

```
#undef VISTAFIER
#define VISTAFIER fsp

class VISTAFIER : public Vistafier {
private:
  var message;
public:
  void Init(int argc, char** argv, boolean base_init=true) {
    if (base_init) Vistafier::Init(argc, argv);
  }
  VISTAFIER(void) {
  }
  VISTAFIER& operator << (const char* s) {
    message += s;
    return self;
  }
  ~VISTAFIER(void) {
    wait_for_children();
    if (message.length() > 0) {
      opipe p("alert");
      p << message;
    }
  }
} VISTAFIER;
```

```
class UnixInteractiveProgram {
private:
  char nm[32];
  char pt[32];
  char pc[32];
  char er[32];
  char qc[32];
public:
  UnixInteractiveProgram(void) {
    nm[0] = pt[0] = pc[0] = er[0] = qc[0] = '\0';
  }
  void Init(const char* exec_dir, const char* name, const char* pause_cmd,
            const char* expected_reply, const char* quit_command) {
    strcpy(nm, name);
    sprintf(pt, "%s/%s", exec_dir, name);
    sprintf(pc, "%s \"%s\"", pause_cmd, expected_reply);
    strcpy(er, expected_reply);
    strcpy(qc, quit_command);
  }
  char* GetName(void) {
    return nm;
  }
  char* GetPath(void) {
    return pt;
  }
  char* GetPauseCommand(void) {
    return pc;
  }
  char* GetExpectedReply(void) {
    return er;
  }
  char* GetQuitCommand(void) {
    return qc;
  }
};
```

```
class TemporaryInputFile {
private:
  char filename[L_tmpnam];
  ifstream ifs;
public:
  TemporaryInputFile(void) {
    tmpnam(filename);
  }
  ~TemporaryInputFile(void) {
    unlink(filename);
  }
  char* GetName(void) {
    return filename;
  }
  ifstream& GetStream(void) {
    int tries = 0, maxtries = 10;
    while (!ifs.is_open() && tries++ < maxtries) {
      ifs.open(filename);
      sleep(1);
    }
    return ifs;
  }
};

class TemporaryOutputFile {
private:
  char filename[L_tmpnam];
  ofstream ofs;
public:
  TemporaryOutputFile(void) {
    tmpnam(filename);
  }
  ~TemporaryOutputFile(void) {
    unlink(filename);
  }
  char* GetName(void) {
    return filename;
  }
  ofstream& GetStream(void) {
    int tries = 0, maxtries = 10;
    while (!ofs.is_open() && tries++ < maxtries) {
      ofs.open(filename);
      sleep(1);
    }
    return ofs;
  }
};
```

```
// Default interval (microseconds) to check for program command completion.
#define DEFAULT_INTERVAL 500000

class UnixInteractiveProgramMaster {
private:
  Pipe ptoc, ctop;
  char rbuf[256];
  UnixInteractiveProgram prog;
  int output_line_count, time, synch_time;
  TemporaryOutputFile output_file;
public:
  UnixInteractiveProgramMaster(void) : output_line_count(0),
    time(DEFAULT_INTERVAL), synch_time(0) {
  }
  int& OutputLineCount(void) {
    return output_line_count;
  }
  UnixInteractiveProgram& Program(void) {
    return prog;
  }
  ofstream& GetOutputFileStream(void) {
    return output_file.GetStream();
  }
  char* GetOutputFileName(void) {
    return output_file.GetName();
  }
  void SetTimer(int t) {
    if (t < 0) time = DEFAULT_INTERVAL; else time = t;
  }
  void SetSynchTime(void) {
    SetTimer(synch_time);
  }
  void PutCommand(char* cmd) {
    ptoc.ofs << cmd << endl;
  }
  void Synch(void) {
    // maintain synchronization by using the pause_cmd to force prog
    // to write out a particular message that Synch waits to see before
    // returning
    if (time == 0) return; // no synchronization
    PutCommand(prog.GetPauseCommand());
    do {
      ctop.ifs >> rbuf;
      if (ctop.ifs.eof() || ctop.ifs.fail()) {
        ctop.ifs.clear();
        usleep(time);
      } else {
        if (strstr(rbuf, prog.GetExpectedReply()))
          break;
      }
    } while (true);
  }
```

```
void SendCommand(char* cmd) {
  PutCommand(cmd);
  Synch();
}
// fork a process running prog (must be called first)
void Init(const char* exec_dir, const char* name, const char* pause_cmd,
          const char* expected_reply, const char* quit_command,
          int synch_time) {
  prog.Init(exec_dir, name, pause_cmd, expected_reply, quit_command);
  this->synch_time = synch_time;
  // set up for parent-to-child (ptoc) and child-to-parent (ctop)
  // communication (two-way in order to keep in synchronization)
  switch (fork()) {
    case -1: {
      syserr("fork");
    }
    case  0: { // child process
      // the output of the ptoc pipe will act as the stdin for prog,
      // similarly, the input of the ctop pipe will act as its stdout
      // (or stderr if prog writes output to stderr)
      if (close(0) == -1)
        syserr("close");
      if (dup(ptoc[INPUT]) != 0)
        syserr("dup");
      if (close(2) == -1)
        syserr("close");
      if (dup(ctop[OUTPUT]) != 2)
        syserr("dup");

      if (close(ptoc[INPUT]) == -1 || close(ptoc[OUTPUT]) == -1 ||
          close(ctop[INPUT]) == -1 || close(ctop[OUTPUT]) == -1)
        syserr("close");
      execlp(prog.GetPath(), prog.GetName(), nil);
      syserr("execlp");
    }
  }
  // parent process closes unneeded pipes
  ptoc.CloseInput();
  ctop.CloseOutput();
  ctop.SetUpForNonBlockingRead();
  Synch();
}
void Pause(const char* dialog = nil, const char* message = nil) {
  if (dialog == nil) {
    printf("Pausing ... Hit <RETURN> to continue.");
    while(getchar() != '\n');
  } else if (message != nil) {
    char syscmd[128];
    sprintf(syscmd, "echo \"%s\" | %s", message, dialog);
    system(syscmd);
  }
}
```

```
  var GetCommand(const char* command) {
    var val = "";
    ipipe ip(command);
    while (ip.Get()) {
      char c = ip.Got();
      val += c;
    }
    return val;
  }
  int Exit(void) {
    PutCommand(prog.GetQuitCommand());
    return 0;
  }
} uipmaster;
```

```
// File: xse.h

#include "vista.h"
#include <xse/event.h>

#undef VISTAFIER
#define VISTAFIER xse

class VISTAFIER : public Vistafier {
private:
  Window wid; // window id (main recipient of X events)
  Window subwid; // sub window id (alternate recipient)
  const char* display;
  boolean treat_strings_as_key_events;
public:
  VISTAFIER(void) : wid(0), subwid(0), display(nil),
                    treat_strings_as_key_events(true) {
  }
  void Init(int argc, char** argv, boolean base_init=true) {
    if (base_init) Vistafier::Init(argc, argv);
    if (argc >= 4) {
      display = argv[1];
      wid = (Window) atol(argv[2]);
      subwid = (Window) atol(argv[3]);
    }
  }
  int GetWid(void) {
    return (int) wid;
  }
  int GetSubWid(void) {
    return (int) subwid;
  }
  VISTAFIER& operator << (int i) {
    treat_strings_as_key_events = !treat_strings_as_key_events;
    return self;
  }
  VISTAFIER& operator << (char c) {
    sendKeyEvent(display, wid, c);
    return self;
  }
  void SendEventString(char* s) {
    if (treat_strings_as_key_events) {
      while (*s)
        self << *s++;
    } else {
      parseAndSendEvents(display, wid, s);
    }
  }
  VISTAFIER& operator << (char* str) {
    SendEventString(str);
    return self;
  }
```

```
  VISTAFIER& operator << (const char* str) {
    SendEventString((char*)str);
    return self;
  }
  VISTAFIER& operator >> (int& x_or_y) {
    static int i=0, x, y;
    if (i%2 == 0) {
      queryPointer(display, subwid, &x, &y);
      x_or_y = x;
    } else {
      x_or_y = y;
    }
    i++;
    return self;
  }
  void ParseEventFile(const char* name) {
    parseFile(display, wid, (char*)name);
  }
  boolean SendNextEvent(void) {
    return (sendNextEvent());
  }
  ~VISTAFIER(void) {
  }
} VISTAFIER;
```

```
// File: unx.h

#include <time.h> // for usleep
#include <strstream.h> // for ostrstream

#include "fsp.h"
#include "xse.h"

#undef VISTAFIER
#define VISTAFIER unx

class VISTAFIER : public Vistafier {
  char evstr[256];
public:
  VISTAFIER(void) {
  }
  void Init(int argc, char** argv, boolean base_init=true) {
    fsp.Init(argc, argv, /* base_init = */ false);
    xse.Init(argc, argv, /* base_init = */ false);
    if (base_init) Vistafier::Init(argc, argv);
  }
  boolean IsBitSet(unsigned long m, int i) {
    unsigned int val = (1 << i);
    return ((m & val) == val);
  }
  boolean IsBitSet(var mods, int i) {
    int bools = mods.strchr('=')+1, boole = mods.strchr(' ');
    unsigned long m = (unsigned long)double(mods(bools,boole-bools+1));
    return (IsBitSet(m, i));
  }
  void SetBooleanOptions(var& command, var& mods,
                         const char* boolean_options[], int len) {
    int bools = mods.strchr('=')+1, boole = mods.strchr(' ');
    unsigned long m = (unsigned long)double(mods(bools,boole-bools+1));
    for (int i = 0; i < len; i++) {
      if (IsBitSet(m, i)) command += boolean_options[i];
    }
  }
  var Find(char* string, var& mods) {
    var result;
    char* r = mods;
    char* p = strstr(r, string);
    if (p != nil) {
      result = p;
    }
    return result;
  }
```

```
  var ExtractStringValue(char* string, var& mods) {
    var stva = Find(string, mods);
    int s = stva.strchr('=');
    char delim = stva[s+1];
    stva = stva(s+2);
    int e = stva.strchr(delim);
    var va = stva(0,e);
    return va;
  }
  var Glob(var& mods) {
    var glob = ExtractStringValue("glob", mods);
    var c("sh -c 'echo ");
    c += glob; c += "'";
    ipipe p(c); c = "";
    while (p.GetWord()) {
      c += " ";
      c += p.GotWord();
    }
    return c;
  }
  const char* ButtonEventString(int b, int d, int x, int y) {
    return sprintf(evstr, "<Btn%d%s> %d %d 0 0 0xffff True %d",
                   b, ((d == 1) ? "Down" : "Up"), x, y, xse.GetSubWid());
  }
  const char* MotionEventString(int n, int x, int y) {
    return sprintf(evstr, "<Motion> %s %d %d 0 0 0xffff True %d",
                   ((n == 1) ? "Normal" : "Hint"), x, y, xse.GetSubWid());
  }
  void ParseEventFile(const char* name) {
    xse.ParseEventFile(name);
  }
  boolean SendNextEvent(void) {
    return xse.SendNextEvent();
  }
  ~VISTAFIER(void) {
  }
} VISTAFIER;
```

```
class acme {
public:
  acme(void) {
  }
  void EnterText(const char* text, int x, int y, boolean overwrite) {
    xse << '0'; // unselects all objects
    xse << 'N'; // name wires
    xse << 0;   // now treat strings as events to parse and send
    xse << unx.ButtonEventString(1, 1, x, y);
    xse << unx.ButtonEventString(1, 0, x, y);
    if (overwrite)
      xse << "c<Key>u"; // select whole text so new entering overwrites old
    else
      xse << "c<Key>e"; // move to end of line so new entering appends to old
    xse << 0;   // back to strings as keyevents
    xse << text;
    xse << 0;
    xse << "<Key>Escape"; // to exit text editing mode
    xse << unx.ButtonEventString(3, 1, x, y);
    xse << unx.ButtonEventString(3, 0, x, y);
    xse << "c<Key>e"; // Not Modified
    xse << 0;
  }
  void EnterLine(int x1, int y1, int x2, int y2, boolean warp_to_end) {
    xse << 'p'; // Port tool
    if (warp_to_end) {
      ostrstream ev; ev << '(' << x2 << ' ' << y2 << ')';
      xse << ev.str(); // warps pointer to x2 y2
    }
    xse << 0;
    xse << unx.ButtonEventString(1, 1, x1, y1);
    xse << unx.ButtonEventString(1, 0, x1, y1);
    xse << unx.ButtonEventString(1, 1, x1, y1);
    xse << unx.ButtonEventString(1, 0, x1, y1);
    xse << unx.MotionEventString(1, x2, y2);
    xse << unx.ButtonEventString(3, 1, x2, y2);
    xse << unx.ButtonEventString(3, 0, x2, y2);
    xse << 0;
  }
  void SelectObject(const char* name, int num = 1, int delay = 100000,
                    boolean clear_wrap = false) {
    for (int i = 0; i < num; i++) {
      if (!clear_wrap || i == 0) {
        xse << '0'; // unselects all objects
        xse << '(' << name << ')'; // selects object by name
      }
      if (clear_wrap) xse << 'k';
      usleep(delay);
      if (clear_wrap) xse << 'k';
    }
    if (clear_wrap) xse << '\005'; // ^E (Not Modified)
  }
```

```
  void SwitchContext(const char* name) {
    xse << 'x'; // Switch Context command key binding
    xse << name;
    xse << 0 << "<Key>Return" << 0;
  }
  boolean ParseWireNameAndValue(char* wirename_equals_value,
                              char* wirename, char* wirevalue) {
    char wnv[40];
    strncpy(wnv, wirename_equals_value, 40);
    char* equals = strchr(wnv, '='); if (equals != nil) *equals = ' ';
    return (sscanf(wnv, "%s %s", wirename, wirevalue) == 2);
  }
  void SelectWarp(const char* object_name) {
    SelectObject(object_name);
    xse << "(Warp Pointer)"; // moves mouse pointer over object
  }
  void ShowWireValue(const char* wire_name, const char* wire_value,
                     boolean overwrite) {
    SelectWarp(wire_name);
    sleep(1); // wait a second for mouse pointer to be warped
    int x, y;
    xse >> x >> y; // gets mouse pointer position
    EnterText(wire_value, x, y+10, overwrite);
  }
  void ShowWireValues(char* values, boolean overwrite) {
    char wn[20], wv[20];
    char *first = strtok(values, " "), *next = nil;
    if (ParseWireNameAndValue(first, wn, wv)) {
      ShowWireValue(wn, wv, overwrite);
    }
    while ((next = strtok(nil, " ")) != nil) {
      if (ParseWireNameAndValue(next, wn, wv)) {
        ShowWireValue(wn, wv, overwrite);
      }
    }
  }
  void EnterTextUnderCell(const char* name, const char* text) {
    ShowWireValue(name, text, true);
  }
  ~acme(void) {
  }
} acme;
```

```
// File: fsm.h

#include "vista.h"
#include <errno.h> // for perror()

#undef VISTAFIER
#define VISTAFIER fsm

class VISTAFIER : public Vistafier {
private:
  const char* sp;
  VarMapIterator vmi;
public:
  VarMap symbols;
  VarMap state_enterings;
  VarMap state_leavings;
  VarMap action_enterings;
  VarMap action_leavings;
  VarMap transitions;
  VarMap states_generated;

  void Init(int argc, char** argv) {
    Vistafier::Init(argc, argv);
  }
  VISTAFIER(void) : sp("  ") {
  }
  void CloseStateLeavings(void) {
    // make sure each state leaving enters itself if it does not
    // already enter another state or action
    vmi(&state_leavings);
    while (vmi++) {
      var key = vmi.key;
      var val = vmi.val;
      if (!state_enterings.element(key) &&
          !action_enterings.element(key)) {
        state_enterings[key] = val(0, val.strchr('.'));
      }
    }
  }
```

```
void TraceTransitions(void) {
  vmi(&state_leavings); // for all state leavings
  while (vmi++) {
    var from = vmi.val; // e.g. "Start"
    var to;
    if (state_enterings.element(vmi.key)) {
      to = state_enterings[vmi.key]; // e.g. "State1"
      transitions.append(from, to);
    } else {
      to = action_enterings[vmi.key];
      var act = to, actions = act;
      VarMapIterator ali;
      boolean done = false;
      while (!done) {
        // find which transition leaves act
        ali(&action_leavings);
        while (ali++) {
          if (ali.val == act)
            break; // ali.key is the one
        }
        if (state_enterings.element(ali.key)) {
          to = state_enterings[ali.key];
          transitions.append(from, to + "/" + actions);
          done = true; // reached a state
        } else {
          to = action_enterings[ali.key];
          act = to; actions += "." + act;
        }
      }
    }
  }
}
void PutStateNameMethod(ofstream& ofs, var sname) {
  sname.format("\"%s\"");
  ofs << "  virtual const char* StateName(void) const {" << endl;
  ofs << "    return " << sname << ";" << endl;
  ofs << "  }" << endl;
}
void ParseAndPutActions(ofstream& ofs, var& does) {
  var act; act.format("%s"); int sep;
  do {
    act = does;
    sep = does.strchr('.');
    if (sep > 0) {
      act = does(0,sep);
      does = does(sep+1);
    }
    if (act.length() > 0) {
      ofs << "    c." << act << "();" << endl;
    }
  } while (sep > 0);
}
```

```
void OpenClassDefinition(ofstream& ofs, const var& sn, const var& curstate) {
  ofs << "class " << sn << curstate;
  ofs << " : public " << sn << " {" << endl;
  ofs << "public:" << endl;
}
void CloseClassDefinition(ofstream& ofs, const var& curstate) {
  // assumes curstate.format("%s");
  ofs << "} " << curstate << "(\"" << curstate << "\");" << endl << endl;
}
void GenerateStateMap(void) {
  var fn(StateFile); fn.format("%s");
  var ab(ActorBase); ab.format("%s");
  var an(ActorName); an.format("%s");
  var sb(StateBase); sb.format("%s");
  var sn(an + sb);   sn.format("%s");
  var start("Start");

  ofstream ofs(fn);
  if (!ofs) {
    cerr << fn;
    perror(" cannot be opened for writing.\n");
  }
  ofs << "// File: " << fn << " (automatically generated -- DO NOT EDIT)";
  ofs << endl << endl;
  ofs << "class " << sn << " : public ";
  ofs << sb << " {" << endl;
  ofs << "public:" << endl;

  // constructor
  ofs << "  " << sn << "(const char* name) {" << endl;
  ofs << "    put_state(this, name);" << endl;
  ofs << "  }" << endl;

  PutStateNameMethod(ofs, sn);

  vmi(&symbols);
  while (vmi++) {
    if (vmi.val == "Transition") {
      ofs << "  virtual void " << vmi.key << "(";
      ofs << an << "& c) {" << endl;
      ofs << "    cerr << c << \"No transition from " << vmi.key;
      ofs << "\" << endl;" << endl << "  }" << endl;
    }
  }
  ofs << "};" << endl << endl;
```

```
    var curstate; curstate.format("%s");
    var curtrans; curtrans.format("%s");
    boolean is_first_state = true;
    vmi(&transitions);
    while (vmi++) {
      int dot = vmi.key.strchr('.');
      boolean is_new_state = (curstate != vmi.key(0, dot));
      if (is_new_state) {
        if (is_first_state) is_first_state = false;
        else CloseClassDefinition(ofs, curstate);
        curstate = vmi.key(0, dot);
        OpenClassDefinition(ofs, sn, curstate);
        PutStateNameMethod(ofs, curstate);
        // keep track of states generated
        states_generated.append(curstate, "yes");
      }
      curtrans = vmi.key(dot+1);
      var goesto(vmi.val); goesto.format("\"%s\"");
      int sep = goesto.strchr('/');
      boolean has_actions = (sep > 0);
      var does; does.format("\"%s\"");
      if (has_actions) {
        does = goesto(sep+1);
        goesto = goesto(0,sep);
      }
      ofs << "  virtual void " << curtrans << "(";
      ofs << ((curtrans == start) ? ab : an) << "& c) {" << endl;
      ofs << "    c.EnterState(";
      ofs << goesto << ");" << endl;
      if (has_actions) {
        ParseAndPutActions(ofs, does);
      }
      ofs << "  }" << endl;
    }
    CloseClassDefinition(ofs, curstate);
    // generate any states that have not yet been generated
    // (these are final states with no transitions)
    vmi(&symbols);
    while (vmi++) {
      if (vmi.val == "State" && !states_generated.element(vmi.key)) {
        OpenClassDefinition(ofs, sn, vmi.key);
        CloseClassDefinition(ofs, vmi.key);
      }
    }
  }
  int Exit(void) {
    CloseStateLeavings();
    TraceTransitions();
    GenerateStateMap();
    return Status();
  }
} VISTAFIER;
```

```
class State : public Cell {
public:
  State(const char* name) : Cell("State", name) {
    VISTAFIER.symbols.append(name, GetType());
  }
};

class Action : public Cell {
public:
  Action(const char* name) : Cell("Action", name) {
    VISTAFIER.symbols.append(name, GetType());
  }
};

class Transition : public Wire {
public:
  Transition(const char* name) : Wire("Transition", name) {
  }
  void EntersAction(const char* name) const {
    VISTAFIER.action_enterings.append(self(), name);
  }
  void LeavesAction(const char* name) {
    VISTAFIER.action_leavings.append(self(), name);
  }
  void EntersState(const char* name) const {
    VISTAFIER.state_enterings.append(self(), name);
  }
  void LeavesState(const char* name) {
    VISTAFIER.state_leavings.append(self(), name);
  }
  void LeavesState(const char* name, const char* type) {
    var nt(name); nt += "."; nt += type;
    VISTAFIER.state_leavings.append(self(), nt);
    if (!VISTAFIER.symbols.element(type)) {
      VISTAFIER.symbols[type] = GetType();
    }
  }
};

#define defState(n, args) \
class n : public State {\
public:\
  n args : State(name) {

#define defAction(n, args) \
class n : public Action {\
public:\
  n args : Action(name) {

#define ENTERS(trans, state) trans.EntersState(state)
#define LEAVES(event, state) event.LeavesState(state, #event)
```

```
// File: cts.h

#include <Dispatch/rpchdr.h>
#include <Dispatch/rpcstream.h>
#ifdef SERVER
#include <Dispatch/rpcreader.h>
#include <Dispatch/rpcservice.h>
#include <unistd.h> // for close()
#endif
#ifdef CLIENT
#include <Dispatch/rpcwriter.h>
#endif
#include <stdlib.h>
#if !defined(CLIENT) && !defined(SERVER)
#include <strstream.h>
#endif

#include "vista.h"
#include "unx.h"

#undef VISTAFIER
#define VISTAFIER cts

class VISTAFIER : public Vistafier {
protected:
  int next_request_number;
  boolean is_client, is_server;
  ofstream ofs;
public:
  int argc;
  char** argv;
  char* Basename(void) {
    char* name = argv[0];
    char* lastslash = strrchr(name, '/');
    if (lastslash != nil) {
      name = lastslash + 1;
    }
    return name;
  }
```

```
  void Init(int argc, char** argv) {
    Vistafier::Init(argc, argv);
    this->argc = argc; this->argv = argv;
    next_request_number = 0;
#if   defined(CLIENT)
    is_client = true;
    state = DONE;
#elif defined(SERVER)
    is_server = true;
    state = DONE;
#else
    unx.Init(argc, argv, /* base_init = */ false);
    is_client = is_server = false;
    char* ofn = getenv("CTSLOG");
    if (ofn != nil && ofn[0] != '\0') {
      ofs.open(ofn);
    } else {
      ostrstream tmp;
      tmp << Basename() << ".1";
      ofn = tmp.str();
      ofs.open(ofn);
      delete [] ofn;
    }
#endif
  }
  VISTAFIER(void) {
  }
  ~VISTAFIER(void) {
  }
  int NextRequestNumber(void) {
    return ++next_request_number;
  }
  boolean IsClient(void) {
    return is_client;
  }
  boolean IsServer(void) {
    return is_server;
  }
  void EnterCellAndWire(Cell& cell, Wire& wire) {
    ofs << "Cell "
        << cell.GetType() << ' '
        << cell.GetName() << ' '
        << wire.GetName() << endl;
  }
#if !defined(CLIENT) && !defined(SERVER)

  VISTAFIER& operator << (Cell& cell) {
    const char* name = cell.GetName();
    acme.SelectWarp(name);
    return self;
  }
```

```
  void EnterMedium(Wire& wire) {
    if (IsAnalyzing()) {
      ofs << "Wire " << wire.GetType() << ' ' << wire.GetName() << ' ';
      ofs << wire << endl;
    }
  }
  void EnterPusher(Cell& cell) {
    if (IsAnalyzing()) {
      ofs << "Cell Pusher "
          << cell.GetType() << ' ' << cell.GetName() << ' ';
    }
  }
  void EnterPusher(Wire& w1, Wire& w2) {
    if (IsAnalyzing()) {
      ofs << w1.GetName() << ' ' << w2.GetName() << endl;
    }
  }
  void EnterPuller(Cell& cell) {
    if (IsAnalyzing()) {
      ofs << "Cell Puller "
          << cell.GetType() << ' ' << cell.GetName() << ' ';
    }
  }
  void EnterPuller(Wire& w1, Wire& w2) {
    if (IsAnalyzing()) {
      // w1 is the ''pullee''
      ofs << w1.GetName() << ' ' << w2.GetName() << ' '
          << w1.GetType() << ' ' << w1 << endl;
    }
  }

  int Exit(void) {
    char* bname = Basename();
    ostrstream tmp;
    tmp << "make -r V=1 OUT=" << bname;
    tmp << ' ' << bname << "client";
    tmp << ' ' << bname << "server";
    tmp << ends;
    char* makecmd = tmp.str();
    int status = system(makecmd);
    delete [] makecmd;
    return status;
  }
#endif
} VISTAFIER;
```

```
class Reader;
class Writer;
class Service;

ostream& check(ostream& os) {
  if (!os.good()) {
    cerr << "ostream eof or failure." << endl;
  }
  return os;
}

#ifdef CLIENT
class Writer : public RpcWriter {
public:
  Writer(const char* path, boolean binary = true)
    : RpcWriter(path, /* fatal = */ true, binary) {
  }
  ~Writer(void) {
  }
  void InitiateRequest(int request_number) {
    server() << RpcHdr(this, request_number);
  }
};
```

```
class ClientMedium : public Wire {
protected:
  Writer* writer;
  rpcstream* client;
  int request_number;
public:
  ClientMedium(const char* type, const char* name)
    : Wire(type, name), writer(nil), client(nil) {
    request_number = VISTAFIER.NextRequestNumber();
  }
  ClientMedium(const char* name)
    : Wire("ClientMedium", name), writer(nil), client(nil) {
  }
  ClientMedium(void) : Wire("ClientMedium", "no_name"),
    writer(nil), client(nil) {
  }
  void CreateWriter(void) {
    writer = new Writer(GetName());
  }
  int GetRequestNumber(void) { return request_number;
  }
  Writer* GetWriter(void) {
    return writer;
  }
  rpcstream& GetClient(void) {
    return *client;
  }
  rpcstream& GetServer(void) {
    return writer->server();
  }
  virtual boolean IsGood(void) { return false;
  }
  boolean IsGood(ClientMedium* cm) {
    if (cm->IsGood()) { // calls the virtual method
      GetWriter()->InitiateRequest(cm->GetRequestNumber());
      return true;
    } else
      return false;
  }
  virtual ~ClientMedium(void) {
    delete writer; writer = nil;
  }
};

class Medium : public ClientMedium {
public:
  Medium(const char* type, const char* name) : ClientMedium(type, name) {}
  Medium(const char* name) : ClientMedium(name) {}
  Medium(void) : ClientMedium() {}
};

#endif
```

```
#ifdef SERVER
class Service : public RpcService {
public:
  Service(const char* path) : RpcService(path) {
  }
  virtual ~Service(void) {
  }
  void Run(void) {
    char* name = VISTAFIER.Basename();
    cout << name << " is open for service at " << _path << endl;
    this->run();
  }
  boolean IsRunning(void) {
    return _running;
  }
  static boolean no_longer_needed;
protected:
  virtual void createReader(int fd);
} *service = nil;

boolean Service::no_longer_needed = false;
```

```
class ServerMedium : public Wire {
protected:
  Reader* reader;
  rpcstream* client;
public:
  ServerMedium(const char* type, const char* name)
    : Wire(type, name), reader(nil), client(nil) {
  }
  ServerMedium(const char* name)
    : Wire("ServerMedium", name), reader(nil), client(nil) {
  }
  ServerMedium(const char* name, Reader* reader, rpcstream* client)
    : Wire("ServerMedium", name),
    reader(reader), client(client) {
  }
  ServerMedium(void) : Wire("ServerMedium", "no_name"),
    reader(nil), client(nil) {
  }
  void CreateService(void) {
    service = new Service(GetName());
  }
  boolean IsReady(void) {
    return (service != nil && service->IsRunning());
  }
  Reader* GetReader(void) {
    return reader;
  }
  rpcstream& GetClient(void) {
    return *client;
  }
  void ServiceNoLongerNeeded(void) {
    Service::no_longer_needed = true;
  }
  virtual ~ServerMedium(void) {
    if (Service::no_longer_needed) {
      delete service; service = nil;
    }
  }
};

class Medium : public ServerMedium {
public:
  Medium(const char* name, Reader* reader, rpcstream* client)
    : ServerMedium(name, reader, client) {
  }
  Medium(const char* type, const char* name) : ServerMedium(type, name) {}
  Medium(const char* name) : ServerMedium("Medium", name) {}
  Medium(void) : ServerMedium() {}
};
#endif
```

```
#if !defined(CLIENT) && !defined(SERVER)

class ClientMedium : virtual public Wire {
public:
  void CreateWriter(void) {
  }
  boolean IsGood(ClientMedium*) {
    return false;
  }
  rpcstream& GetServer(void) {
    return *(new rpcstream);
  }
};

class ServerMedium : virtual public Wire {
public:
  void CreateService(void) {
  }
  boolean IsReady(void) {
    return false;
  }
  rpcstream& GetClient(void) {
    return *(new rpcstream);
  }
  void ServiceNoLongerNeeded(void) {
  }
};

class Medium : public ClientMedium, public ServerMedium {
protected:
  int request_number;
public:
  Medium(const char* type, const char* name) {
    request_number = VISTAFIER.NextRequestNumber();
    Init(type, name, request_number);
    VISTAFIER.EnterMedium(self);
  }
  Medium(const char* name) {
    request_number = 0;
    Init("Medium", "0", name);
    VISTAFIER.EnterMedium(self);
  }
  Medium(void) {
  }
};

#endif
```

```
// File: ptp.h

#ifdef PEER
#include <Dispatch/dispatcher.h>
#include <Dispatch/rpchdr.h>
#include <Dispatch/rpcpeer.h>
#include <Dispatch/rpcreader.h>
#include <Dispatch/rpcwriter.h>
#include <unistd.h> // for close()
#endif
#include <Dispatch/rpcstream.h>
#include <stdlib.h>
#include <strstream.h>

#include "vista.h"

#undef VISTAFIER
#define VISTAFIER ptp

class VISTAFIER : public Vistafier {
protected:
  int next_request_number;
  ofstream ofs;
public:
  int argc;
  char** argv;
  char* Basename(void) {
    char* name = argv[0];
    char* lastslash = strrchr(name, '/');
    if (lastslash != nil) {
      name = lastslash + 1;
    }
    return name;
  }
  void Init(int argc, char** argv) {
    Vistafier::Init(argc, argv);
    this->argc = argc; this->argv = argv;
    next_request_number = 0;
#ifdef PEER
    state = DONE;
#else
    char* ofn = getenv("CTSLOG");
    if (ofn != nil && ofn[0] != '\0') {
      ofs.open(ofn);
    } else {
      ostrstream tmp;
      tmp << Basename() << ".1";
      ofn = tmp.str();
      ofs.open(ofn);
      delete [] ofn;
    }
#endif
```

```
  }
  VISTAFIER(void) {
  }
  ~VISTAFIER(void) {
  }
  int NextRequestNumber(void) {
    return ++next_request_number;
  }
#ifndef PEER
  void EnterMedium(Wire& wire) {
    if (IsAnalyzing()) {
      ofs << "Wire " << wire.GetType() << ' ' << wire.GetName() << ' ';
      ofs << wire << endl;
    }
  }
  int Exit(void) {
    char* bname = Basename();
    ostrstream tmp;
    tmp << "make -r V=1 OUT=" << bname;
    tmp << ' ' << bname << "peer";
    tmp << ends;
    char* makecmd = tmp.str();
    int status = system(makecmd);
    delete [] makecmd;
    return status;
  }
#endif
} VISTAFIER;

class Reader; class Writer;

ostream& check(ostream& os) {
  if (!os.good()) {
    cerr << "ostream eof or failure." << endl;
  }
  return os;
}

#ifdef PEER

class Writer : public RpcWriter {
public:
  Writer(const char* host, int port, boolean binary = true)
    : RpcWriter(host, port, /* fatal = */ false, binary) {
  }
  Writer(int fd, boolean binary = true)
    : RpcWriter(fd, /* fatal = */ false, binary) {
  }
  void InitiateRequest(int request_number) {
    server() << RpcHdr(this, request_number);
  }
};
```

```
class Peer : public RpcPeer {
protected:
  Writer* writer; Reader* reader;
public:
  static boolean local_done, remote_done;
  // If the peer is running, open a connection to it.  If not, create a
  // socket so that the peer can open a connection to us.
  Peer(const char* path)
    : RpcPeer(path), writer(nil), reader(nil) {
    init(path);
  }
  virtual ~Peer(void);
  boolean IsRunning(void) {
    return _running;
  }
  void Run(void) {
    if (!IsRunning()) {
      run();
    }
  }
  void Stop(void) {
    if (IsRunning()) {
      quitRunning();
    }
  }
  boolean ShouldRun(void) {
    if (!local_done || !remote_done) {
      Dispatcher::instance().dispatch();
    }
    return (!local_done || !remote_done);
  }
  Writer* GetWriter(void) {
    return writer;
  }
  Reader* GetReader(void) {
    return reader;
  }
protected:
  // Open a connection to the peer's socket.
  virtual boolean createReaderAndWriter(const char* rHost, int rPort);
  // The peer just opened a connection to our socket.
  virtual void createReaderAndWriter(int fd);
  void Error(int fd, int) {
    cerr << "Peer: can't talk to more than one peer." << endl;
    close(fd);
    local_done = remote_done = true;
  }
  void Error(const char* host, int port) {
    cerr << "Peer: error opening connection to peer at ";
    cerr << host << '.' << port << endl;
    delete writer; writer = nil;
```

```
    local_done = remote_done = true;
  }
  void Error(int fd) {
    cerr << "Peer: error accepting connection from peer on " << fd << endl;
    delete writer; writer = nil;
    local_done = remote_done = true;
  }
};

boolean Peer::local_done = false;
boolean Peer::remote_done = false;

class Medium : public Wire {
protected:
  Peer* peer;
  Reader* reader;
  Writer* writer;
  rpcstream* client;
  int request_number;
public:
  Medium(const char* type, const char* name)
    : Wire(type, name),
    peer(nil), reader(nil), writer(nil), client(nil) {
      request_number = VISTAFIER.NextRequestNumber();
  }
  Medium(const char* name)
    : Wire("Medium", name),
    peer(nil), reader(nil), writer(nil), client(nil) {
  }
  Medium(const char* name, Reader* reader, rpcstream* client)
    : Wire("Medium", name),
    peer(nil), reader(reader), writer(nil), client(client) {
  }
  Medium(void) : Wire("Medium", "no_name"),
    peer(nil), reader(nil), writer(nil), client(nil) {
  }
  Peer* GetPeer(void) {
    return peer;
  }
  Reader* GetReader(void) {
    return reader;
  }
  Writer* GetWriter(void) {
    return writer;
  }
```

```
  rpcstream& GetStream(void) {
    if (client != nil) {
      return *client;
    }
    return writer->server();
  }
  int GetRequestNumber(void) {
    return request_number;
  }
  virtual boolean IsGood(void) {
    return false;
  }
  boolean IsGood(Medium* m) {
    if (m->IsGood()) { // calls the virtual method
      if (peer == nil) {
        peer = new Peer(GetName());
      }
      if ((writer = peer->GetWriter()) != nil) {
        writer->InitiateRequest(m->GetRequestNumber());
        return true;
      }
    }
    return false;
  }
  boolean IsReady(void) {
    return (client != nil);
  }
  boolean PeerShouldRun(void) {
    return (peer != nil && peer->ShouldRun());
  }
  void Done(void) {
    if (client == nil) {
      Peer::local_done = true;
    } else {
      Peer::remote_done = true;
    }
  }
  virtual ~Medium(void) {
    delete peer;
  }
};
```

```
#else

class Medium : public Wire {
protected:
  int request_number;
public:
  Medium(const char* type, const char* name) {
    request_number = VISTAFIER.NextRequestNumber();
    Init(type, name, request_number);
    VISTAFIER.EnterMedium(self);
  }
  Medium(const char* name) {
    request_number = 0;
    Init("Medium", "0", name);
  }
  Medium(void) {
  }
  virtual boolean IsGood(void) {
    return false;
  }
  boolean IsGood(Medium*) {
    return false;
  }
  boolean IsReady(void) {
    return false;
  }
  boolean PeerShouldRun(void) {
    return false;
  }
  rpcstream& GetStream(void) {
    return *(new rpcstream);
  }
  void Done(void) {
  }
};
#endif
```

```
// File: sse.h

#include "unx.h"
#include "ptp.h"

class SpreadSheet {
protected:
  int col, row; // current cell position
  int l, b, r, t; // current region (left, bottom, right, top)
  char buf[80]; // for sprintf formatting of commands
  char curregion[16]; // current region in string form
public:
  var name;
  SpreadSheet(const char* name) : name(name), col(1), row(1) {
    l = b = r = t = 1;
  }
  virtual void SelectCell(int col, int row) {
  }
  virtual void PutNumericValue(double) {
  }
  virtual void PutStringValue(char*) {
  }
  virtual void PutNumericFormula(char*) {
  }
  virtual void PutStringFormula(char*) {
  }
  virtual void Quit(void) {
  }
  virtual void WriteAndReadRegion(void) {
  }
  virtual void SetCurrentRegion(void) {
  }
  void SetCurrentRegion(int lft, int bot, int rgt, int top) {
    l = lft; b = bot; r = rgt; t = top;
    SetCurrentRegion(); // the virtual method
  }
  double GetNumericValue(void) {
    WriteAndReadRegion();
    return atof(buf);
  }
  char* GetStringValue(void) {
    WriteAndReadRegion();
    return strdup(buf);
  }
  int GetCol(void) {
    return col;
  }
  int GetRow(void) {
    return row;
  }
  void GetColRow(int& C, int& R) { C = col; R = row;
  }
```

```
  int SetCol(int C) {
    int prev = col; col = C; return prev;
  }
  int SetRow(int R) {
    int prev = row; row = R; return prev;
  }
  void SetColRow(int C, int R) {
    col = C; row = R;
  }
};

class Sc : public SpreadSheet {
public:
  Sc(void) : SpreadSheet("sc") {}
  void SetCurrentRegion(void) {
    if (l == r && b == t) {
      sprintf(curregion, "%c%d", r-1+'a', b-1);
    } else {
      sprintf(curregion, "%c%d:%c%d", l-1+'a', t-1, r-1+'a', b-1);
    }
  }
  void SelectCell(int x, int y) {
    sprintf(buf, "g%c%d\n", x-1+'a', y-1);
    xse << buf;
  }
  void PutNumericValue(double d) {
    sprintf(buf, "x=%g\n", d);
    xse << buf;
  }
  void PutStringValue(char* s) {
    sprintf(buf, "x<%s\n", s);
    xse << buf;
  }
  void PutNumericFormula(char* f) {
    sprintf(buf, "x=@%s(%s)\n", f, curregion);
    xse << buf;
  }
  void PutStringFormula(char* f) {
    sprintf(buf, "xER@%s\n", f);
    xse << buf;
  }
  void WriteAndReadRegion(void) {
    TemporaryInputFile tif;
    sprintf(buf, "W%s\" %s\n", tif.GetName(), curregion);
    xse << buf;
    tif.GetStream().getline(buf, sizeof(buf), '\n');
  }
  void Quit(void) {
    xse << 'q' << 'n'; // Do you want a chance to save the data? (No)
  }
};
```

```
class Oleo : public SpreadSheet {
public:
  Oleo(void) : SpreadSheet("oleo") {}
  void SetCurrentRegion(void) {
    if (l == r && b == t) {
      sprintf(curregion, "r%dc%d", t, l);
    } else {
      sprintf(curregion, "r%d:%dc%d:%d", t, b, l, r);
    }
  }
  void DeleteCellContents(void) {
    xse << 0 << "<key>Delete" << 0;
  }
  void SendCommand(void) {
    xse << 0 << "<key>Return" << 0;
  }
  void SelectCell(int x, int y) {
    sprintf(buf, "%cjr%dc%d", '\030' /*^X*/, y, x);
    xse << buf;
    SendCommand();
  }
  void PutNumericValue(double d) {
    DeleteCellContents();
    sprintf(buf, "%g", d);
    xse << buf;
    SendCommand();
  }
  void PutStringValue(char* s) {
    DeleteCellContents();
    xse << 0 << "s<key>quotedbl" << 0 << s << 0 << "s<key>quotedbl" << 0;
    SendCommand();
  }
  void PutNumericFormula(char* f) {
    sprintf(buf, "%s(%s)", f, curregion);
    DeleteCellContents();
    xse << buf;
    SendCommand();
  }
  void PutStringFormula(char* f) {
    DeleteCellContents();
    xse << f;
    SendCommand();
  }
  void WriteAndReadRegion(void) {
    TemporaryInputFile tif;
    sprintf(buf, "%c%ca%s", '\033' /*ESC*/, '\020' /*^P*/, curregion);
    xse << buf;
    SendCommand();
    xse << tif.GetName();
    SendCommand();
    tif.GetStream().getline(buf, sizeof(buf), '\n');
  }
```

```
  void Quit(void) {
    xse << 0 << "c<key>x" << "c<key>c" << 0;
    // Spreadsheet modified.  Quit without saving? (yes or no)
    xse << "yes";
    SendCommand();
  }
};

#undef VISTAFIER
#define VISTAFIER sse

class VISTAFIER : public Vistafier
#ifdef PEER
, public IOHandler {
#else
  {
#endif
private:
  char cmd;
public:
  SpreadSheet* spreadsheet;
  VISTAFIER(void) : spreadsheet(nil), cmd('x') {
  }
  void Init(int argc, char** argv) {
    Vistafier::Init(argc, argv);
    ptp.Init(argc, argv, /* base_init = */ false);
    if (argc == 2 && ptp.IsDone()) {
      var ssname = argv[1];
      ipipe p(ssname);
      if (p.GetWord()) {
        var xid(p.GotWord());
        char** av = new char*[4];
        av[0] = argv[0];
        av[1] = ":0.0";
        av[2] = STR xid;
        av[3] = "0";
        unx.Init(4, av, /* base_init = */ false);
        delete [] av;
        if (ssname == "oleo") spreadsheet = new Oleo;
        else if (ssname == "sc") spreadsheet = new Sc;
#ifdef PEER
        Dispatcher::instance().link(0, Dispatcher::ReadMask, this);
#endif
      } else {
        state = DONE;
      }
    } else {
      unx.Init(argc, argv, /* base_init = */ false);
    }
  }
```

```
  int Exit(void) {
    return ptp.Exit();
  }
  void Quit(void) {
    spreadsheet->Quit();
    cout << spreadsheet->name << " is quitting." << endl;
    state = DONE;
#ifdef PEER
    Peer::local_done = Peer::remote_done = true;
#endif
  }
  int inputReady(int fd) {
    cin >> cmd;
    return 0; // call select before getting more input
  }
  boolean GotCommand(const char* name) {
    if (cmd == name[0]) { cmd = '\0'; return true; } else return false;
  }
  ~VISTAFIER(void) {
    delete spreadsheet;
  }
} VISTAFIER;
```

APPENDIX B

SUPPORTING CODE

```
void Editor::Vistafy(void) {
  const char* name = GetCellMatrixName();
  if (name == nil || !OfferToSave()) return;
  const char* oldp = vistadialog->Text();
  char* parameters = vistadialog->Edit(oldp == nil ? "." : nil);
  if (parameters == nil) return;
  boolean vista_button_states[4];
  vistadialog->GetButtonStates(vista_button_states);
  boolean code_it = vista_button_states[0],
          make_it = vista_button_states[1],
          show_it = vista_button_states[2],
          run_it  = vista_button_states[3];
  char* ewb = acme->ErrorWarnBuf();
  char* cmfilname = get_cellmatrix_filename(name, true);
  char* fullname = GetProtoLib()->GetFileName(name);
  const char* plibname = GetProtoLib()->GetName();
  const char* plibfullpath = GetProtoLib()->GetFullPath();
  char* cmtxfullpath = GetProtoLib()->GetFileName(cmfilname);
  char* dotcfullpath = get_filename_with_suffix(fullname, ".c");
  time_t dotcfiletime = get_mod_time(dotcfullpath);
  time_t cmtxfiletime = get_mod_time(cmtxfullpath);
  if (cmtxfiletime > dotcfiletime && code_it) {
    WriteSIMFile(VISTA); // update dotc file
    cellmatrix->WriteDependencies(".d");
  }
  boolean out_of_date = false, ok_to_make = false, ok_to_run = false;
  char makecmd[255], runcmd[255];
  sprintf(makecmd, "make -q -r -C %s OUT=%s V=0", plibfullpath, name);
  // see if executable is up-to-date (-q => query) and if not, make it
  if (make_it) {
    int status = system(makecmd);
    ok_to_make = out_of_date = (status != 0);
  }
  if (ok_to_make) {
    sprintf(makecmd, "make -r -C %s OUT=%s V=%d", plibfullpath, name, show_it);
    int pid = fork();
    if (pid == -1) {
      sprintf(ewb, "Cannot fork to make %s", name);
      Complain(ewb);
    } else if (pid == 0) { // child process
```

```
      set_child_process_group();
      errno = 0;
      int status = system(makecmd);
      if (status != 0) {
        sprintf(ewb, "Making \"%s\" failed with status %d, errno %d",
                name, status, errno);
        acme->ErrorWarnMessage(ewb);
      }
      _exit(status);
    } else {
      // parent process
      sprintf(ewb, "Making \"%s\" in \"%s\"", name, plibname);
      if (acme->AbortProcess(pid, ewb)) {
        // killing the make process group does *not* always remove
        // a partially written executable, so do it here, ignoring
        // the error if the executable file does not exist
        unlink(fullname);
      }
    }
  }
  if (show_it) {
    // manifest problems discovered by analyzer during make, namely
    // 1. incompatibly-typed ports connected
    // 2. ports with no default parameters not connected
    boolean incompatibly_typed_ports_connected = false;
    boolean some_cells_need_connections = false;
    CellList cells_needing_connections;
    // cells_needing_connections non-empty => these cells have ports with
    // no defined default parameters, so they need to have wires attached
    char* dotzfullpath = get_filename_with_suffix(fullname, ".z");
    ifstream dotz(dotzfullpath); char buf[40];
    while (dotz) {
      dotz >> buf;
      char* ampersand = strchr(buf, '&');
      if (ampersand != nil) {
        incompatibly_typed_ports_connected = true;
      } else {
        Cell* cell = cellmatrix->FindCell(buf, /*only_if_denominated=*/false);
        if (cell != nil) {
          cells_needing_connections.Append(new_(CellNode(cell)));
        }
      }
    }
    if (incompatibly_typed_ports_connected) {
      Complain("There are incompatibly-typed ports connected.");
      FindTypedWires();
    }
    some_cells_need_connections = (cells_needing_connections.Size() > 0);
    if (some_cells_need_connections) {
      cellmatrix->Select(&cells_needing_connections,
                         /* visibly = */ true, /* extend = */ false);
      if (cells_needing_connections.Size() == 1) {
```

```
        Complain("The selected cell has one or more connections missing.");
      } else {
        Complain("The selected cells have one or more connections missing.");
      }
    }
    if (incompatibly_typed_ports_connected || some_cells_need_connections) {
      run_it = false;
    }
  }
  if (run_it) {
    sprintf(makecmd, "make -q -r -C %s OUT=%s V=0", plibfullpath, name);
    int status = system(makecmd);
    ok_to_run = (!out_of_date || (status == 0));
  }
  if (ok_to_run) {
    // up-to-date, so fork/exec it
    signal(SIGHUP, SIG_IGN); signal(SIGINT, SIG_IGN); signal(SIGQUIT, SIG_IGN);
    int pid = fork();
    if (pid == -1) {
      sprintf(ewb, "Cannot fork to exec %s", name);
      Complain(ewb);
    } else if (pid == 0) { // child process
      set_child_process_group();
      char* disp = acme->GetDisplayName();
      boolean use_system = ((strchr(parameters, '<') != nil) ||
                            (strchr(parameters, '>') != nil) ||
                            (strchr(parameters, '|') != nil));
      if (use_system) {
        sprintf(runcmd, "%s %s %d %d %d %s", fullname, disp,
                acme->GetWindowId(), cellmatrixview->GetWindowId(),
                show_it, parameters);
        int status = system(runcmd); _exit(status);
      } else {
        // use execl
        const char* vl = "VISTALOG"; const char* sx = ".a";
        char* vle = new_(char[strlen(vl)+strlen(name)+strlen(sx)+2]); // yes, 2
        strcpy(vle,vl); strcat(vle,"="); strcat(vle,name); strcat(vle,sx);
        putenv(vle);
        char awid[16]; sprintf(awid, "%d", acme->GetWindowId());
        char cwid[16]; sprintf(cwid, "%d", cellmatrixview->GetWindowId());
        char show[16]; sprintf(show, "%d", show_it);
        execl(fullname, name, disp, awid, cwid, show, parameters, (char*)nil);
        syserr("execl");
      }
    } else {
      // parent process needs to know the pid of the child that was just
      // forked off in order to acknowledge receipt of signals from it
      acme->SetAckPid(pid);
      acme->SetSleepTime(dump_depth * 10000);
    }
  }
}
```

The following is an English description of the Vistafy command, enumerating the steps necessary to produce, analyze and run an executable Vista specification. Each of the four subsystems has its own subsequent subsection, even though, as mentioned above, these four modules are very interrelated. Refer to Figure B.2, which shows the dialogue box presented to the user upon invocation of the Vistafy command. This is the second of three dialogue boxes the user sees, (the third is found in Figure B.3,) but it only pops up if the current cellmatrix design has a name, and either no changes need to be saved to a disk file, or else the user refuses the offer to save a modified design before proceeding. In the case of an unsaved modified design, the dialogue box shown in Figure B.1 will appear.

The cellmatrix name is used as the base name for several ancillary and intermediate files that are produced by the Vista subsystems, as well as the final result, which is the executable agent. This executable C++ program has the identical name as its source cellmatrix, that is, no file extension is appended to it. The parameters entered in the string-editor box are passed on to this executable program. By convention, the first parameter is the pathname of the directory the program will connect to before it begins its business. This pathname is "." initially, meaning the current working directory of the environment in which Acme was invoked. The parameters entered are saved, and in subsequent Vistafy invocations (in the same Acme session) are restored from their saved state, for continuity and convenience. The user can abort the Vistafy command either by blanking out all parameter entries, or by clicking on the "pushbutton" labeled *Cancel*. Otherwise, pressing the *Return* key or pushing the *OK* button tells Vista to proceed.

The four vertically-aligned buttons are "checkboxes" that are originally checked, by default. Clicking on a checkbox toggles its state from checked to unchecked and back again. The checked state is indicated visually by the "X" in the box, the unchecked state by a blank box. The ramifications of these four button states are related below. Four boolean variables, *code_it*, *make_it*, *show_it* and *run_it* are defined, taking their initial values (true or false) from the states of the buttons labeled *Code*, *Make*, *Show* and *Run*, respectively. In the following outline of the remaining steps, assume that `test` names the user-created cellmatrix design, and `trial` names the (user-defined) working protolib, which must reference the base *vista* protolib, and which contains the `test` design:

1. Look for a file named `test.c` in the `trial` protolib directory. If it does not exist, or is out of date with respect to the `test.cm` cellmatrix file; then, if *code_it* is also true, generate the `test.c` file by invoking the Codifier (see Subsection 3.4.1), and, in addition, create a `test.d` file containing the file dependencies for `test.c` that **make** will consult in step 2.

2. If *make_it* is true, issue a Unix shell command (via the *system* call) to `make -q OUT=test`. The `-q` option tells **make** to "query" only, that is, to determine by examining the last-modified times of each file the target `test` depends on if the `test` executable is up to date or not, but if not, *not* to try to make it.

3. If not up to date, make the `test` executable, again by calling `make OUT=test` (minus the query flag,) only this time, use the Unix *fork* call to create a subprocess in which to do it.

4. While the child process is running `make OUT=test`, the Acme parent process pops up the final dialogue box. Figure B.3 shows its appearance for this test example. This dialogue box goes away by itself when the child process exits, unless the user pushes the *Abort* button, which both dismisses the dialogue box and terminates the child process prematurely.

5. If the make process terminates normally, and if *show_it* is true, there will be a file named `test.z` in the `trial` directory. This is the output file created by the Analyzer, which is invoked by the make. See Subsection 3.4.2.

6. If *show_it* is true, and the `test.z` file exists (and is non-empty,) the Manifester will examine this file and report to the user any problems therein recorded. See Subsection 3.4.3 and the paragraphs below.

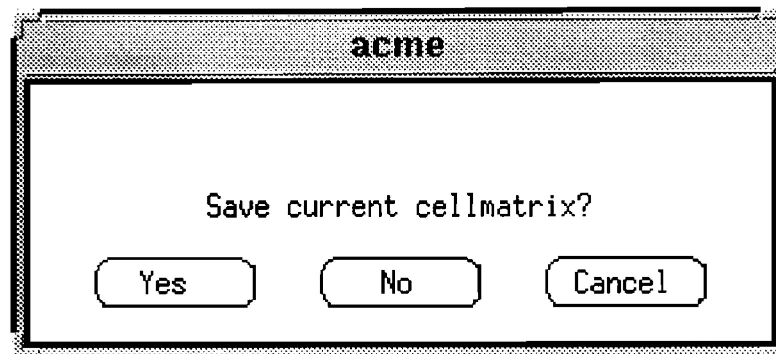


Figure B.1. The "Offer to Save" Dialogue

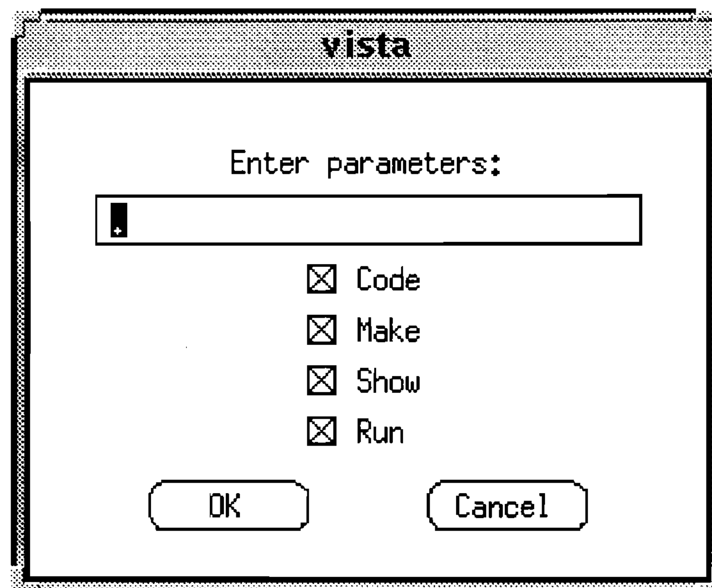


Figure B.2. The Vistafy Command Dialogue

Figure B.3. Abort Making "test" in "trial" Dialogue

7. If *run_it* is true, using *system* once more "shell out" a make query, and if the return status is zero (meaning up to date,) set another boolean variable *ok_to_run* to be true, otherwise set it false. If the Analyzer detected syntax errors, the executable will not have been created (or updated), and therefore cannot be run.

8. If *ok_to_run* is true, using the Unix *fork/exec* system-call pair, create a child process to execute the **test** specification. See Subsection 3.4.4 for details. In the parent process, save for subsequent reference by Acme the *process id* of the child process that was just forked off. The need for this is explained in Subsection 4.3.3.

Recall that in Chapter 3, Subsection 3.4.3, what the Manifester does was described. The details were omitted there, but given here.

If non-empty, there are only two kinds of entries in the **.z** file produced by the Analyzer. An entry will either contain an ampersand character (**&**) or else it will not. Its presence indicates incompatibly-typed ports were connected, the ampersand being the special marker used to separate the port type names, as indicated in Subsection 3.4.2. Its absence thus implies that this entry names a cell missing one or more needed connections. Hence, an ambiguity is possible should the cell name contain an ampersand; however, Acme does not complain about such cell names, because cell names can be any string of characters, as there are no corresponding C++ identifiers, as is the case for wire names.

During its processing scan of the **.z** file, the Manifester conditionally sets a boolean variable, *incompatibly_typed_ports_connected*, and appends each cell (found by name lookup in the cellmatrix) to a list of cells needing connections. After the scan, if *incompatibly_typed_ports_connected* is true, a notification box will pop up as shown in Figure B.4. After the user dismisses this notification by pushing the *OK* button, the Manifester invokes the *FindTypedWires* command automatically. This Acme command, which the user *could* invoke manually before trying the Vistafy command, queries the top-level cellmatrix only and performs the same type inferencing that was done by the Analyzer during the netlist generation, as mentioned in Subsection 3.4.1. It then pops up a string-editor/string-chooser dialogue box, as shown in Figure B.5, allowing the user to choose, and Acme to then select and highlight the badly-typed wires.

If the list mentioned in the preceding paragraph is *non*-empty, then each cell in the list is selected and highlighted, and the notification box shown in Figure B.6 appears. The user must then inspect each selected cell, and supply the missing connections. If either there are cells thus selected, or the boolean variable *incompatibly_typed_ports_connected* is true, then the boolean *run_it* is set false, forestalling the final Executer steps 7 and 8 outlined above.

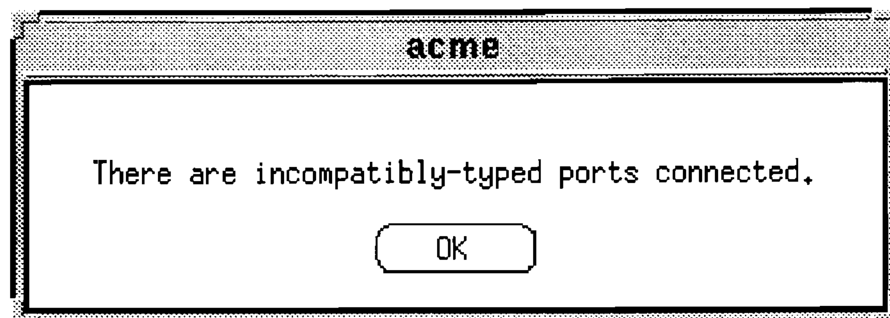


Figure B.4. The Incompatibility Complaint Notification

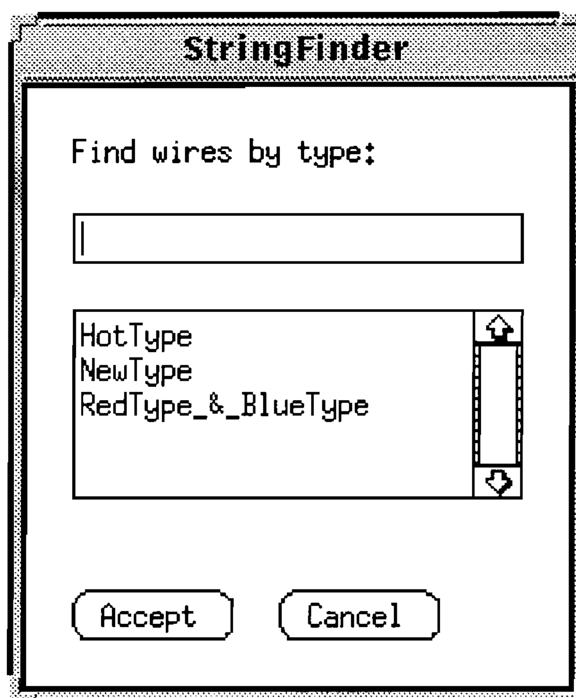


Figure B.5. The "Find Typed Wires" Finder Dialogue

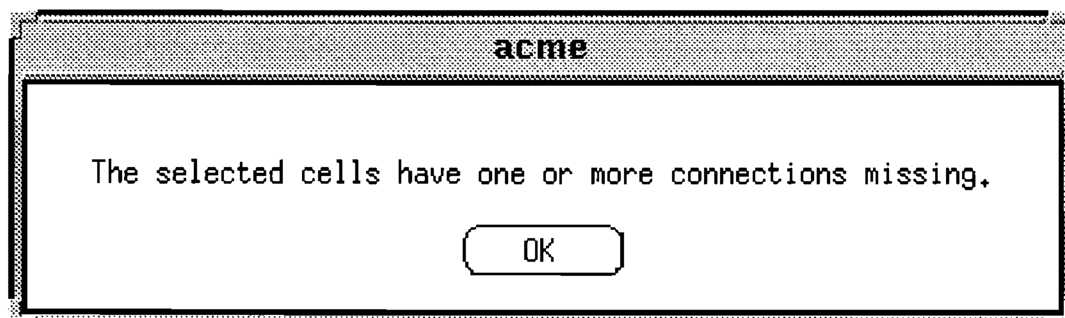


Figure B.6. The Missing Connections Complaint Notification

```
// File: dispatch.c

#include <Dispatch/dispatcher.h>
#include <Dispatch/iohandler.h>
#include <fcntl.h>
#include <fstream.h>
#include <stdlib.h>

extern int errno;
static const int MYBUFSIZE = 128;
Dispatcher* dp;
boolean broadcast_redistribute(char*, int);
#                                                          ifdef ORDERED_JOINING
int reader_order(void), reader_turn(void);
#                                                          endif
class MyIOHandler : public IOHandler {
private:
  int fd;
  fstream fs;
  Dispatcher::DispatcherMask dpmask;
  char buf[MYBUFSIZE];
  int bufsize;
#                                                          ifdef ORDERED_JOINING
  int order;
#                                                          endif
public:
  boolean IsReader(void) {
    return (dpmask == Dispatcher::ReadMask);
  }
  boolean IsWriter(void) {
    return (dpmask == Dispatcher::WriteMask);
  }
  boolean IsLinked(void) {
    return (this == dp->handler(fd, dpmask));
  }
#                                                          ifdef ORDERED_JOINING
  void ReLink(void) {
    dp->link(fd, dpmask, this);
  }
#                                                          endif
  void SetBlocking(boolean on) {
    // turn blocking on or off
    int fflags = fcntl(fd, F_GETFL, 0);
    int    blockflags = fflags & ~O_NDELAY; // make sure O_NDELAY is off
    int nonblockflags = fflags |  O_NDELAY; // make sure O_NDELAY is on
    fcntl(fd, F_SETFL, on ? blockflags : nonblockflags);
  }
  MyIOHandler(int fd, boolean is_reader)
    : fd(fd), fs(fd), bufsize(MYBUFSIZE),
    dpmask(is_reader ? Dispatcher::ReadMask : Dispatcher::WriteMask) {
    if (is_reader) {
      SetBlocking(false);
```

```
#                                                           ifdef ORDERED_JOINING
      order = reader_order();
#                                                           endif
    }
    buf[0] = EOF;
  }
  ~MyIOHandler(void) {
  }
  virtual int inputReady(int fd) {
    int retval = 0; // call select before getting more input
    errno = 0; // may be EWOULDBLOCK after while loop below
    int num_lines_got = 0;
#                                                           ifdef ORDERED_JOINING
    if (order != reader_turn()) {
      return -1; // must wait turn
    }
#                                                           endif
    while (fs.getline(buf, bufsize)) {
      if (!broadcast_redistribute(buf, ++num_lines_got)) {
        break;
      }
    }
    if (!fs.eof()) {
      retval = 1; // didn't get all input it could, don't call select yet
    } else if (fs.eof()) {
      fs.clear();
      if (num_lines_got == 0 && errno == 0) {
        // eof and errno == EWOULDBLOCK means a *temporarily* empty pipe
        // eof and errno == 0 means *permanently* empty, as its writer has died
        retval = -1; // error or does not want to read anything more
      }
    } else if (fs.fail()) {
      retval = -1;
    } else {
      retval = -1; // unknown stream state
    }
    return retval;
  }
  virtual int outputReady(int fd) {
    int retval = -1;
    if (buf[0] != EOF) {
      fs << buf << endl;
      buf[0] = EOF;
      retval = -1;
    }
    return retval;
  }
  void Write(char* buf) {
    fs << buf << endl;
  }
};
```

```
MyIOHandler** handlers;
int nreaders, nwriters, nhandlers;
#                                                    ifdef ORDERED_JOINING
int reader_order_, reader_turn_;

int reader_order(void) { return reader_order_++; }

int reader_turn(void) { return (reader_turn_ % nreaders); }

void relink_readers(void) {
  for (int i = 0; i < nhandlers; i++) {
    if (handlers[i]->IsReader() && !handlers[i]->IsLinked()) {
      handlers[i]->ReLink();
    }
  }
}
#                                                    endif
boolean broadcast_redistribute(char* buf, int nlines) {
  for (int i = 0; i < nhandlers; i++) {
    if (handlers[i]->IsWriter()) {
      handlers[i]->Write(buf);
    }
  }
  if (nreaders > 1 && nlines == 1) {
#                                                    ifdef ORDERED_JOINING
    reader_turn_++;
    relink_readers();
#                                                    endif
    return false; // multiple readers only get one line at a time
  } else {
    return true;
  }
}

boolean any_readers_alive(void) {
  for (int i = 0; i < nhandlers; i++) {
    if (handlers[i]->IsReader() && handlers[i]->IsLinked()) {
      return true;
    }
  }
  return false;
}

int main(int argc, char** argv) {
  dp = &Dispatcher::instance();
  nhandlers = argc-2; nreaders = nwriters = 0;
#                                                    ifdef ORDERED_JOINING
  reader_order_ = reader_turn_ = 0;
#                                                    endif
  handlers = new MyIOHandler*[nhandlers];
  for (int i = 0; i < nhandlers; i++) {
    int arg = atoi(argv[i+2]);
```

```
    boolean reader = (arg >= 0);
    int fd = (reader) ? arg : -arg;
    dp->link(fd, (reader) ? Dispatcher::ReadMask : Dispatcher::WriteMask,
             handlers[i] = new MyIOHandler(fd, reader));
    if (reader) nreaders++; else nwriters++;
  }
  while (any_readers_alive()) {
    // block indefinitely until an I/O condition occurs
    dp->dispatch();
  }
  for (i = 0; i < nhandlers; i++) {
    delete handlers[i];
  }
  delete [] handlers;
  delete dp;
  return 0;
}

/////////////////////////////////////////////////////////////////////////////////

// File: xsekey.c

#include <stdio.h>
extern char* charToStr();

int main () {
  char c;
  while((c=getchar())!=EOF)
    puts(charToStr(c));
  return 0;
}
```

```
// File: dollars.c

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

const char* EnglishOnes[] = {
  "one", "two", "three", "four", "five", "six", "seven", "eight", "nine"
};

const char* EnglishTens[] = {
  "", "twenty", "thirty", "forty", "fifty", "sixty", "seventy",
  "eighty", "ninety"
};

const char* EnglishTeens[] = {
  "ten", "eleven", "twelve", "thirteen", "fourteen", "fifteen",
  "sixteen", "seventeen", "eighteen", "nineteen"
};

const char* EnglishPeriods[] = {
  " ", " thousand", " million", " billion", " trillion", " quadrillion",
  " sextillion", " septillion", " octillion", " nonillion", " decillion"
};

void english_small_cardinal(char* buf, int n) {
  int hundreds = n/100;
  int rem = n % 100;
  if (hundreds > 0) {
    strcat(buf, EnglishOnes[hundreds-1]);
    strcat(buf, " hundred");
    if (rem > 0) {
      strcat(buf, " ");
    }
  }
  if (rem > 0) {
    int tens = rem/10;
    int ones = n % 10;
    if (tens > 1) {
      strcat(buf, EnglishTens[tens-1]);
      if (ones > 0) {
        strcat(buf, "-");
        strcat(buf, EnglishOnes[ones-1]);
      }
    } else if (tens == 1) {
      strcat(buf, EnglishTeens[ones]);
    } else if (ones > 0) {
      strcat(buf, EnglishOnes[ones-1]);
    }
  }
}
```

```
void english_cardinal(char* buf, int n, int period) {
  int beyond = int(n/1000);
  int here = n % 1000;
  if (period > 10) {
    strcat(buf, "Number too large to print in English: ");
    char tmp[10];
    strcat(buf, sprintf(tmp, "%d", n));
    return;
  }
  if (beyond != 0) {
    english_cardinal(buf, beyond, period+1);
  }
  if (here != 0) {
    if (beyond != 0) {
      strcat(buf, " ");
    }
    english_small_cardinal(buf, here);
    strcat(buf, EnglishPeriods[period]);
  }
}

char* dollars(double d) {
  char result[1024]; result[0] = '\0';
  long c = irint((d - long(d)) * 100);
  long n = long(d);
  char tmp[10], tmp1[10], cents[12];
  sprintf(tmp1, (c == 0 ? "no" : sprintf(tmp, "%2d", c)));
  sprintf(cents, "and %s/100", tmp1);
  if (n < 0) {
    strcat(result, "negative ");
    english_cardinal(result, -n, 0);
  } else if (n == 0) {
    strcat(result, "zero ");
  } else {
    english_cardinal(result, n, 0);
  }
  strcat(result, cents);
  return strdup(result);
}

int main(int argc, char** argv) {
  if (argc != 2) {
    cerr << "Usage: " << argv[0] << " <dollar_amount>" << endl;
    exit(1);
  }
  double d = atof(argv[1]);
  char* s = dollars(d);
  cout << s << endl;
  delete [] s;
  return 0;
}
```

```
#!/bin/csh -f
#
# File: cmnt2code
#
# Purpose: compute a comments-to-code ratio for acme's authors
#
set stripper = /home/grad/neff/acme/vista/libs/fsm/stripper
set awkfile  = /tmp/cmnt2code.awk
cat << EOF > $awkfile
function get_author(fn, authors) {
  auth = "????"; tmp = "awk '/Author/{print \$3}' " fn;
  fn_sans_suffix = substr(fn, 1, index(fn, "."));
  if (tmp | getline auth > 0) {
    close(tmp);
    authors[fn_sans_suffix] = auth;
  } else {
    if (fn_sans_suffix in authors) auth = authors[fn_sans_suffix];
  }
  return auth;
}
function get_stripped_wc(fn) {
  wc = 0; tmp = "$stripper " fn;
  if (tmp | getline wc > 0) close(tmp);
  return wc;
}
\$3 !~ /total/ && \$3 !~ /firgen/ {
  auth = get_author(\$3, authors); wc = get_stripped_wc(\$3);
  if (auth in totfiles) {
    totfiles[auth] ++;
    totlines[auth] += \$1;
    totbytes[auth] += \$2;
    tot_code[auth] += wc;
  } else {
    totfiles[auth] = 1;
    totlines[auth] = \$1;
    totbytes[auth] = \$2;
    tot_code[auth] = wc;
  }
}
END {
  printf "\n%s%s\n\n", "    Total Files   Total Lines   Total Bytes",
                      "   % Comments    % Code       Ratio";
  gtotfiles = gtotlines = gtotbytes = 0;
  for (auth in totfiles) {
    gtotfiles += totfiles[auth];
    gtotlines += totlines[auth];
    gtotbytes += totbytes[auth];
    codebytes = tot_code[auth];
    cmntbytes = totbytes[auth] - codebytes;
    cmnt2code = cmntbytes / (codebytes == 0 ? 1 : codebytes);
    prct_cmnt = cmnt2code / (1 + cmnt2code);
    prct_code = 1 / (1 + cmnt2code);
```

```
    printf "%4s: %7d %12d %12d %12.2f %12.2f %12.2f\n",\
           auth, totfiles[auth], totlines[auth], totbytes[auth],\
           prct_cmnt, prct_code, cmnt2code;
  }
  printf "\n%13d %12d %12d\n", gtotfiles, gtotlines, gtotbytes;
}
EOF

wc -lc /home/grad/neff/acme/sources/*/*/*.{c,h,inlines,orgc}\
      | sort +2 | gawk -f $awkfile; rm $awkfile
```

```
#!/usr/vlsi/tcl/bin/tcl -f
#
# File: tclc2c
#
# Purpose: compute a comments-to-code ratio for acme's authors
#

proc get_author {file} {
  set auth [exec awk /Author/ $file]
  if {$auth != ""} { return [lindex $auth 2] }
  set file [format "%s.h" [file rootname $file]]
  if [file exists $file] {
    set auth [exec awk /Author/ $file]
  }
  if {$auth != ""} { return [lindex $auth 2] }
  return "????"
}

proc skip_file {file} {
 return {[string first "firgen" $file] > 0}
}

set acmefiles [glob /home/grad/neff/acme/sources/*/*/*.{c,h,orgc,inlines}]
set stripper /home/grad/neff/acme/vista/libs/fsm/stripper

foreach file $acmefiles {
  if [skip_file $file] continue
  set linebytecount [exec wc -lc < $file]
  set linecount [lindex $linebytecount 0]
  set bytecount [lindex $linebytecount 1]
  set codecount [exec $stripper $file]
  set auth [get_author $file]
  if [info exists totfiles($auth)] {
    incr totfiles($auth)
    incr totlines($auth) $linecount
    incr totbytes($auth) $bytecount
    incr tot_code($auth) $codecount
  } else {
    set totfiles($auth) 1
    set totlines($auth) $linecount
    set totbytes($auth) $bytecount
    set tot_code($auth) $codecount
  }
}

puts stdout [format "\n%s%s\n"\
                    "    Total Files   Total Lines   Total Bytes"\
                    "   % Comments    % Code        Ratio"]
set gtotfiles 0
set gtotlines 0
set gtotbytes 0
```

```
foreach auth [array names totfiles] {
  incr gtotfiles $totfiles($auth)
  incr gtotlines $totlines($auth)
  incr gtotbytes $totbytes($auth)
  set codebytes $tot_code($auth)
  set cmntbytes [expr $totbytes($auth)-$codebytes]
  set cmnt2code [expr $cmntbytes/($codebytes==0?1.:$codebytes.)]
  set prct_cmnt [expr $cmnt2code/(1.+$cmnt2code)]
  set prct_code [expr 1./(1.+$cmnt2code)]
  puts stdout [format "%4s: %7d %12d %12d %12.2f %12.2f %12.2f"\
              $auth $totfiles($auth) $totlines($auth) $totbytes($auth)\
              $prct_cmnt $prct_code $cmnt2code]
}

puts stdout [format "\n%13d %12d %12d" $gtotfiles $gtotlines $gtotbytes]
```

APPENDIX C

MAKEFILES

```
# File: Makefile.common
#
# Common (shared) makefile for vista (VISTA = .)
#
CC = g++
ifeq ($V,1)
DV = -DVERBOSE
endif
DEBUGFLAGS = -g
EXTRAFLAGS = -finline -finline-functions -Winline -pipe
STDINCLUDES = -I$(VISTA) -I/usr/vlsi/local/lib/g++-include
INCLUDES = -I. $(STDINCLUDES)
CCFLAGS = $(DEBUGFLAGS) $(EXTRAFLAGS) $(INCLUDES) $(DV)
LIBPATHS = -L. -L$(VISTA) -L$(VISTA)/var -L/usr/openwin/lib
LIBNAMES = -lvista -lvar -lX11 -lm
LINKFLAGS = $(LIBPATHS) $(LIBNAMES)
ifdef HAS_CELLS
HAS_CELLS_OR_WIRES = 1
endif
ifdef HAS_WIRES
HAS_CELLS_OR_WIRES = 1
endif
PROTOLIB = $(shell basename $(shell pwd))

.SUFFIXES: .o .c

.c.o:
	@$(CC) -c $< $(CCFLAGS)

$(OUT):
ifeq ($V,1)
	@echo Making \"$(OUT)\" in \"$(PROTOLIB)\" ...
	@rm -f $(OUT).z; $(MAKE) analysis
	@test -s $(OUT).z || $(CC) -o $(OUT) $(OUT).c $(CCFLAGS) $(LINKFLAGS)
	@test -x $(OUT) && echo done making.
else
```

```
        @$(CC) -o $(OUT) $(OUT).c $(CCFLAGS) $(LINKFLAGS)
        $(OUTADDL)
endif

analysis: $(OUT).z
# analyzer checks for incompatible types and missing default parameters
        @echo done analyzing.

$(OUT).z:
        @awk '$$1 ~ /_&_/ {print $$1}' $(OUT).c | sort | uniq > $(OUT).z
        @grep :: $(OUT).c | cut -d, -f2- | sed -e 's/,//g' -e 's/);//g' |\
        tr ' ' '\012' | sort | uniq | grep :: | sed 's/:://g' > $(OUT).n
        @$(CC) -E $(CCFLAGS) $(OUT).c | awk '/^} vv/, /;$$/' > $(OUT).l
        @xargs -n1 -iz awk '$$0 ~ /z/{print "z"}' $(OUT).l < $(OUT).n >$(OUT).y
        @diff $(OUT).n $(OUT).y | awk '$$1 == "<" { print $$2}' |\
        xargs -n1 -iz grep ::z $(OUT).c | cut -d\" -f2 | sort | uniq >>$(OUT).z
        @rm -f $(OUT).n $(OUT).l $(OUT).y

manifest: $(OUT).q
# manifester calls coral (and Explain) to ascertain if $(OUT) is okay
        @coral -q < $(OUT).q > $(OUT).r
        @grep -s "Number of Answers = 1" $(OUT).r
        @echo done manifesting.

$(OUT).q: $(OUT).P $(PROTOLIB).q
        @echo "/* File: $@ */" > $@
        @echo "consult($(OUT).P)." >> $@
        @echo "explain_on." >> $@
        @cat $(PROTOLIB).q >> $@
        @echo "explain_off." >> $@
        @echo "shell(\"Explain -f dump_directory > /dev/null 2>&1\")." >> $@
        @echo "quit." >> $@

$(OUT).P: $(OUT).F $(PROTOLIB).P
        @echo "/* File: $@ */" > $@
        @echo "module $(OUT)." >> $@
        @echo "" >> $@
        @cat $(OUT).F $(PROTOLIB).P >> $@
        @echo "" >> $@
        @echo "end_module." >> $@

$(OUT).F: $(OUT).a $(VISTA)/factify.el
        @emacs -batch -q $(shell /bin/pwd)/$@ -l $(VISTA)/factify.el \
        -f save-buffer -kill >/dev/null

# $(OUT).a gets created by $(OUT) when it is executed.
```

```
$(VISTA)/factify.el:
	@grep "^#:" Makefile.common | cut -d: -f2- > $@

#:(setq make-backup-files nil)
#:(setq fname (file-name-nondirectory (buffer-file-name)))
#:(let ((file (concat (substring fname 0 (or (string-match "\\.F" fname) 0))
#:                    ".a")))
#:  (if (file-readable-p file)
#:    (insert-file-contents file)))
#:(goto-char (point-min))
#:(if (search-forward "Vistafier")
#:  (progn (beginning-of-line) (kill-line)
#:         (insert (concat "/* File: " fname " */"))))
#:(while (not (looking-at "========"))
#:  (progn (downcase-word 1) (end-of-line) (insert ".") (forward-char 1)))
#:(delete-region (point) (point-max))

doth::
ifdef CELL
	@echo Making $(CELL).h
	@echo "// File: "$(CELL)".h" > $(CELL).h
	@echo "" >> $(CELL).h
ifdef HAS_WIRES
	@echo "#define BEGIN_"$(CELL) >> $(CELL).h
endif
ifdef HAS_CELLS
	@echo "#define BEGIN_"$(CELL)"_CELLS" >> $(CELL).h
	@echo "#define END_"$(CELL)"_CELLS" >> $(CELL).h
endif
ifdef HAS_WIRES
	@echo "#define END_"$(CELL) >> $(CELL).h
endif
ifdef HAS_CELLS_OR_WIRES
	@echo "" >> $(CELL).h
endif
	@echo -n "defCell("$(CELL)"," >> $(CELL).h
	@grep $(CELL).h $(OUT).c | awk -F/ '{ printf "%s", $$3}' >> $(CELL).h
	@echo ")" >> $(CELL).h
else

ifdef WIRE
	@echo Making $(WIRE).h
	@echo "// File: "$(WIRE)".h" > $(WIRE).h
	@echo "" >> $(WIRE).h
	@echo "#ifndef "$(WIRE)"_h" >> $(WIRE).h
```

```
	@echo "#define "$(WIRE)"_h" >> $(WIRE).h
	@echo "" >> $(WIRE).h
	@echo "class "$(WIRE)" : public Wire {" >> $(WIRE).h
	@echo "public:" >> $(WIRE).h
	@echo "  "$(WIRE)"(const char* name) : Wire(\""$(WIRE)"\", name) {" \
	>> $(WIRE).h
	@echo "  }" >> $(WIRE).h
	@echo "  ~"$(WIRE)"(void) {" >> $(WIRE).h
	@echo "  }" >> $(WIRE).h
	@echo "};" >> $(WIRE).h
	@echo "" >> $(WIRE).h
	@echo "#endif" >> $(WIRE).h
else
	@echo Making $(OUT).h
	@echo "// File: "$(OUT)".h" > $(OUT).h
	@echo "" >> $(OUT).h
	@echo "#define BEGIN MAIN_BEGIN" >> $(OUT).h
	@echo "" >> $(OUT).h
	@echo "#define BEGIN_CELLS MAIN_BEGIN_CELLS" >> $(OUT).h
	@echo "" >> $(OUT).h
	@echo "#define END_CELLS MAIN_END_CELLS" >> $(OUT).h
	@echo "" >> $(OUT).h
	@echo "#define END MAIN_END" >> $(OUT).h
endif
endif

each:
	@for each in $(foreach f, $(wildcard *.c), $(basename $f)); \
	do echo $$each; $(MAKE) OUT=$$each; done

clean::
	rm -f $(OUT) core
```

```
###############################################
# Makefile: sig vista
###############################################

VISTA = ../vista

include $(VISTA)/Makefile.common

include $(OUT).d

###############################################
# Makefile: fsp vista
###############################################

VISTA = ../vista

include $(VISTA)/Makefile.common

include $(OUT).d

###############################################
# Makefile: xse vista
###############################################

VISTA = ../vista

include $(VISTA)/Makefile.common

LIBPATHS := $(LIBPATHS) -L$(VISTA)/xse
LIBNAMES := -lxse $(LIBNAMES)

include $(OUT).d

xsekey: xsekey.c
	@cc -O -o xsekey xsekey.c $(LIBPATHS) $(LIBNAMES)

###############################################
# Makefile: unx fsp xse vista
###############################################

VISTA = ../vista

include $(VISTA)/Makefile.common

IVINCLUD = -I/usr/vlsi/interviews/include
INCLUDES = -I. -I../fsp -I../xse $(IVINCLUD) $(STDINCLUDES)
```

```
IVLIBPTH = /usr/vlsi/interviews/lib/SUN4.debug
LIBPATHS := $(LIBPATHS) -L$(VISTA)/xse -L$(IVLIBPTH)
LIBNAMES := -lxse -lIV $(LIBNAMES)

include $(OUT).d
```

```
##############################################
# Makefile: fsm vista
##############################################

VISTA = ../vista

OUTADDL = $(OUT) && $(MAKE) FSM

include $(VISTA)/Makefile.common

INCLUDES := $(INCLUDES) -I$(VISTA)/fsm

LIBPATHS := $(LIBPATHS) -L$(VISTA)/fsm
LIBNAMES := $(LIBNAMES) -lfsm

FSM: $(OUT)FSM.c $(OUT)FSM.h
	$(CC) -o $(OUT) $(OUT)FSM.c $(CCFLAGS) $(LINKFLAGS)

include $(OUT).d
```

```
###############################################
# Makefile: cts (client to server) vista
###############################################

VISTA = ../vista

include $(VISTA)/Makefile.common

AUTOGEN = (automatically generated -- DO NOT EDIT)
DV = -Dcplusplus_2_1
IVINCLUD = -I/usr/vlsi/interviews/include
INCLUDES = -I. -I../unx -I../fsp -I../xse $(IVINCLUD) $(STDINCLUDES)
IVLIBPTH = /usr/vlsi/interviews/lib/SUN4.debug
LIBPATHS := $(LIBPATHS) -L$(VISTA)/xse -L$(IVLIBPTH)
LIBNAMES := -lxse -lIV $(LIBNAMES)

include $(OUT).d

$(OUT).1: $(OUT)
	$(OUT)

$(OUT).2: $(OUT).1
	grep "Wire Medium" < $< | cut -d" " -f4 > $@

$(OUT).3: $(OUT).1
	awk '$$1 == "Wire" && $$2 != "Medium"' < $<  > $@

$(OUT).4: $(OUT).3
	cut -d" " -f4 < $< > $@

$(OUT).5: $(OUT).1
	grep Pusher < $< | cut -d" " -f3 > $@

$(OUT).6: $(OUT).1
	grep Puller < $< | cut -d" " -f3 > $@

$(OUT).7: $(OUT).1
	awk '$$1=="Cell" && $$2=="Puller"\
	{printf "\n#define CELL%d %s\n#define WIRE%d %s\n",\
	$$8, $$3, $$8, $$7; numfn++}\
	END {printf "\n#define NUMFN %d\n\n", numfn}\
	' < $< > $@

$(OUT)client.h: $(OUT).h
	sed \
	-e 's/.h/client.h $(AUTOGEN)/'\
```

```
        -e 's/ MAIN_BEGIN_CELLS//'\
        -e 's/ MAIN_END_CELLS//'\
        < $< > $@
        /usr/5bin/echo "\n#define CLIENT" >> $@

$(OUT)server.h: $(OUT).h
        sed \
        -e 's/.h/server.h $(AUTOGEN)/'\
        -e 's/ MAIN_BEGIN_CELLS//'\
        -e 's/ MAIN_END_CELLS/ service->Run();/'\
        < $< > $@
        /usr/5bin/echo "\n#define SERVER" >> $@

$(OUT)client.c: $(OUT)client.h

$(OUT)client.c: $(OUT).5 $(OUT).3 $(OUT).2 $(OUT).1
        /usr/5bin/echo "// File: $(OUT)client.c $(AUTOGEN)\n" > $@
        echo "#include \"$(OUT)client.h\"" >> $@
        /usr/5bin/echo "#include \"$(PROTOLIB).h\"\n" >> $@
        /usr/5bin/echo "#include \"Client.h\"\noutclude" >> $@
        xargs -n1 -ip /usr/5bin/echo "#include \"p.h\"\noutclude" \
        < $(OUT).5 >> $@
        /usr/5bin/echo "\nBEGIN\n" >> $@
        xargs -n1 -ip echo "     Medium vv0(\"p\");" < $(OUT).2 >> $@
        awk '\
        {printf "     %s vv%s(\"%s\");\n", $$2, $$3, $$3} \
        ' $(OUT).3 >> $@
        /usr/5bin/echo "\nBEGIN_CELLS\n" >> $@
        awk '\
        /Client/\
        {printf "    %s(\"%s\", vv%d);\n", $$2, $$3, $$4} \
        /Pusher/\
        {printf "    %s(\"%s\", vv%d, vv%d);\n", $$3, $$4, $$5, $$6} \
        ' $(OUT).1 >> $@
        /usr/5bin/echo "\nEND_CELLS\n\nEND" >> $@

$(OUT)server.c: $(OUT)server.h $(OUT)reader.h

$(OUT)server.c: $(OUT).6 $(OUT).3 $(OUT).2 $(OUT).1
        /usr/5bin/echo "// File: $(OUT)server.c $(AUTOGEN)\n" > $@
        echo "#include \"$(OUT)server.h\"" >> $@
        /usr/5bin/echo "#include \"$(PROTOLIB).h\"\n" >> $@
        /usr/5bin/echo "#include \"Server.h\"\noutclude" >> $@
        xargs -n1 -ip /usr/5bin/echo "#include \"p.h\"\noutclude" \
        < $(OUT).6 >> $@
        /usr/5bin/echo "\n#include \"$(OUT)reader.h\"" >> $@
```

```
        /usr/5bin/echo "\nBEGIN\n" >> $@
        xargs -n1 -ip echo "     Medium vv0(\"p\");" < $(OUT).2 >> $@
        awk '\
        {printf "     %s vv%s(\"%s\");\n", $$2, $$3, $$3} \
        ' $(OUT).3 >> $@
        /usr/5bin/echo "\nBEGIN_CELLS\n" >> $@
        awk '\
        /Server/\
        {printf "    %s(\"%s\", vv%d);\n", $$2, $$3, $$4} \
        ' $(OUT).1 >> $@
        awk '\
        /Puller/\
        {printf "    %s(\"%s\", vv%d, vv%d);\n", $$3, $$4, $$5, $$6} \
        ' $(OUT).1 >> $@
        /usr/5bin/echo "\nEND_CELLS\n\nEND" >> $@

$(OUT)reader.h: $(OUT).7 $(OUT).4 Makefile
        echo "// File: $@ $(AUTOGEN)" > $@
        cat $(OUT).7 >> $@
        grep "^#1" Makefile | cut -d1 -f2- >> $@
        xargs -n1 -in echo "  READ("n")" < $(OUT).4 >> $@
        grep "^#2" Makefile | cut -d2 -f2- >> $@
        xargs -n1 -iz echo "    _function["z"] = &Reader::read"z";" \
        < $(OUT).4 >> $@
        grep "^#3" Makefile | cut -d3 -f2- >> $@

$(OUT)client: $(OUT)client.c $(PROTOLIB).h
        $(CC) -o $@ $(OUT)client.c gmalloc.o $(CCFLAGS) $(LINKFLAGS)

$(OUT)server: $(OUT)server.c $(PROTOLIB).h
        $(CC) -o $@ $(OUT)server.c gmalloc.o $(CCFLAGS) $(LINKFLAGS)

#1#define READ(N) \
#1  static void read##N(RpcReader* reader, \
#1                      RpcHdr& hdr,rpcstream& client){\
#1    Medium mdm("read"#N,(Reader*)reader, &client); \
#1    WIRE##N arg(hdr.ndata()); \
#1    CELL##N("read"#N, arg, mdm); \
#1  }
#1
#1class Reader : public RpcReader {
#1protected:
#1  RpcService* _service;
#1  virtual void connectionClosed(int fd) {
#1    close(fd);
#1    if (Service::no_longer_needed) {
```

```
#1      _service->quitRunning();
#1    }
#1    delete this;
#1  }
#2public:
#2  Reader(int fd, RpcService* service)
#2    : RpcReader(fd, NUMFN + 1, /* binary = */ true),
#2    _service(service) {
#2    client().setf(ios::dont_close);
#3  }
#3  virtual ~Reader(void) {
#3  }
#3};
#3
#3void Service::createReader(int fd) {
#3  new Reader(fd, this);
#3}
```

```
##############################################
# Makefile: ptp (peer to peer) vista
##############################################

VISTA = ../vista

include $(VISTA)/Makefile.common

AUTOGEN = (automatically generated -- DO NOT EDIT)
DV = -Dcplusplus_2_1
IVINCLUD = -I/usr/vlsi/interviews/include
INCLUDES = -I. -I../unx -I../fsp -I../xse $(IVINCLUD) $(STDINCLUDES)
IVLIBPTH = /usr/vlsi/interviews/lib/SUN4.debug
LIBPATHS := $(LIBPATHS) -L$(VISTA)/xse -L$(IVLIBPTH)
LIBNAMES := -lxse -lIV $(LIBNAMES)

include $(OUT).d

$(OUT).1:
	$(OUT)

$(OUT).2: $(OUT).c
	awk '$$1 == "Medium" { print $$2}' < $< | cut -d"(" -f1 > $@

$(OUT).3: $(OUT).1
	awk '$$1 == "Wire"' < $< | sort +1 > $@

$(OUT).4: $(OUT).3
	cut -d" " -f4 < $< > $@

$(OUT).5: $(OUT).c $(OUT).3
	awk '$$4 == "(NAME," && $$5 != "BIDIR(Medium," \
	{ print substr($$2,2,length($$2)-4), substr($$5,7,length($$5)-7) }' \
	< $(OUT).c | sort +1 | join -j 2 -o 1.1 1.2 2.4 -t" " - $(OUT).3 > $@

$(OUT).6: $(OUT).5
	awk '{printf "\n#define CELL%d %s\n#define WIRE%d %s\n",\
	$$3, $$1, $$3, $$2; numfn++}\
	END {printf "\n#define NUMFN %d\n\n", numfn}\
	' < $< > $@

$(OUT)peer.h: $(OUT).h
	sed \
	-e 's/$(OUT).h/$(OUT)peer.h $(AUTOGEN)/'\
	-e 's/MAIN_BEGIN_CELLS/do {/'\
	-e 's/MAIN_END_CELLS/} while (vv0.PeerShouldRun());/'\
```

```
        < $< > $@
        /usr/5bin/echo "\n#define PEER" >> $@

$(OUT)reader.h: $(OUT).6 $(OUT).4 Makefile
        echo "// File: $@ $(AUTOGEN)" > $@
        cat $(OUT).6 >> $@
        grep "^#1" < Makefile | cut -d1 -f2- >> $@
        xargs -n1 -in echo "  READ("n")" < $(OUT).4 >> $@
        grep "^#2" Makefile | cut -d2 -f2- >> $@
        xargs -n1 -iz echo "    _function["z"] = &Reader::read"z";" \
        < $(OUT).4 >> $@
        grep "^#3" < Makefile | cut -d3 -f2- >> $@

$(OUT)peer.c: $(OUT)peer.h $(OUT)reader.h $(PROTOLIB).h

$(OUT)peer.c: $(OUT).2 $(OUT).c
        sed \
        -e 's/$(OUT).c/$(OUT)peer.c $(AUTOGEN)/'\
        -e 's/$(OUT).h/$(OUT)peer.h/'\
        -e 's/$(shell cat $(OUT).2)/vv0/g'\
        < $(OUT).c |\
        awk '$$1 == "BEGIN" {printf "#include \"$(OUT)reader.h\"\n\n%s\n",$$1}\
             $$1 != "BEGIN" {print}' > $@

$(OUT)peer: $(OUT)peer.c
        $(CC) -o $@ $(OUT)peer.c gmalloc.o $(CCFLAGS) $(LINKFLAGS)

#1#define READ(N) \
#1  static void read##N(RpcReader* reader, \
#1                      RpcHdr& hdr,rpcstream& client){\
#1    Medium mdm("read"#N,(Reader*)reader, &client); \
#1    WIRE##N arg(hdr.ndata()); \
#1    CELL##N("read"#N, arg, mdm); \
#1  }
#1
#1class Reader : public RpcReader {
#1protected:
#1  Peer* peer;
#1  virtual void connectionClosed(int fd) {
#1#ifdef CLOSESTOP
#1    close(fd);
#1    peer->Stop();
#1#endif
#1  }
#2public:
#2  Reader(rpcstream* client, Peer* peer)
```

```
#2    : RpcReader(client, NUMFN + 1),
#2    peer(peer) {
#2    client->setf(ios::dont_close);
#3  }
#3};
#3
#3virtual Peer::~Peer(void) {
#3  delete reader;
#3  delete writer;
#3}
#3
#3virtual boolean Peer::createReaderAndWriter
#3                (const char* rHost, int rPort) {
#3  writer = new Writer(rHost, rPort);
#3  if (writer->server()) {
#3    reader = new Reader(&writer->server(), this);
#3    return true;
#3  } else {
#3    Error(rHost, rPort);
#3    return false;
#3  }
#3}
#3
#3virtual void Peer::createReaderAndWriter(int fd) {
#3  if (writer) {
#3    Error(fd, 0);
#3    return;
#3  }
#3  writer = new Writer(fd);
#3  if (writer->server()) {
#3    reader = new Reader(&writer->server(), this);
#3  } else {
#3    Error(fd);
#3  }
#3}
```

```
# File: sim.mk
#
# Makefile to run simppl (and spplice, if necessary)
#
ACME_WINDOW_ID = $(shell xwininfo -name acme | awk '/xwininfo/{print $$5}')
NAME = exor
SIM = simppl
OPT = -c
CMD = $(SIM) $(NAME)
GEO = -geometry 80x40+0+0
I=0
# interactive I=1
C=0
# copy output C=1

.SUFFIXES: .ppx.ppl.cm

%.ppx : %.ppl
	@spplice $* > /dev/null

%.ppl : %.cm
	@echo Making $@ from $<
	@xse -window $(ACME_WINDOW_ID) 's<Key>l' '<Key>Return'

$(NAME): $(NAME).ppx $(NAME).src
ifeq ($I,0)
	@$(SIM) $(OPT) $(NAME) < $(NAME).src
else
	@toolwait xterm $(GEO) -xrm "*allowSendEvents:true" -e $(CMD)
	@echo "" | alert button="Click here to begin simulation"
ifeq ($C,1)
	@xse -window $(SIM) '<Key>c' '<Key>o' '<Key>p' '<Key>Return'
endif
	@xse -window $(SIM) '<Key>s' '<Key>o' '<Key>Return'
	@echo "When ready to exit $(SIM)" | alert button="click here"
	@xse -window $(SIM) '<Key>e' '<Key>Return'
endif
	@rm $(NAME).src

$(NAME).src:
	@echo "options -echo"  > $(NAME).src
	@for i in $(INPUTS); do\
	 if echo $$i | grep -s ':';\
	 then echo "set" $$i;\
	 else echo $$i; echo "show" $(WATCH);\
	 fi; done >> $(NAME).src
```

```
# File: adsim.mk
#
# Makefile to do mixed analog/digital design simulation using viewlogic
# (madsnet and madssim) and hspice
#
# NAME comes from the adsim script that invokes make with this Makefile
# (e.g., NAME=adtest)
#
PVNAME = 'Powerview Cockpit'
PVDIR = $(shell echo $(WDIR) | cut -d: -f1)
PROJDIR = $(PVDIR)/$(NAME)
PROJECT_LIST = $(PVDIR)/vf/project.lst
SPPLICE = spplice -V -f
MADSNET = madsnet
MADSSIM = madssim
MADSSIMGEO = -geometry 80x66+0+0
SPICELINK_SWITCHES = $(HOME)/spicelink_switches.ini
# hscale args: label width_in_pixels from to tickinterval initialvalue
TICKSIZEP = $(shell hscale "TICKSIZE (seconds, power of)" 500 -15 -1 1 -10)
TICKSIZET = $(shell echo $(TICKSIZEP) | tee ticksize.tmp | sed 's/-/3k10 12 /')
TICKSIZE  = $(shell echo "$(TICKSIZET)-^p" | dc)
SSMIN = $(shell awk '{print 1+$$1}' ticksize.tmp)
STEPSIZEP = $(shell hscale "STEPSIZE (seconds, power of)" 500 $(SSMIN) 0 1 -7)
STEPSIZET = $(shell echo $(STEPSIZEP) | sed 's/-/2k10 12 /')
STEPSIZE  = $(shell echo "$(STEPSIZET)-^p" | dc)

all: .setup goal .cleanup

.setup: $(NAME).ppl $(NAME).src $(SPICELINK_SWITCHES) set-project

goal: $(PROJDIR)/$(NAME).wfm
	cd $(PROJDIR); viewtrace $(NAME).wfm

$(NAME).ppl:
	@echo "After writing $(NAME).ppl" | alert button="click here"

$(NAME).src:
	@echo "After writing $(NAME).src" | alert button="click here"

$(SPICELINK_SWITCHES):
	@echo -mnfxv > $(SPICELINK_SWITCHES)

$(NAME).cir: $(NAME).1 $(NAME).cmd viewsim.ini
	cp -p $(NAME).1 $(PROJDIR)/wir
	# add digital watch list to $(NAME).cmd
	echo -n "watch " > $(NAME).tmp
```

```
	grep "^wave" $(NAME).cmd | cut -d" " -f3- >> $(NAME).tmp
	sed '/^wave/r $(NAME).tmp' < $(NAME).cmd |\
	sed 's/QUIT/ECHO Reached end of $(NAME).cmd file: enter q to quit./'\
	> $(NAME)-cmd.tmp; mv $(NAME)-cmd.tmp $(NAME).cmd
	cp -p $(NAME).cmd viewsim.ini $(PROJDIR)
	cd $(PROJDIR); $(MADSNET) $(NAME) > /dev/null 2>&1
	mv $(PROJDIR)/$(NAME).cir .

$(PROJDIR)/$(NAME).wfm: $(PROJDIR)/$(NAME).cir
	cd $(PROJDIR); \
	xterm -T $(MADSSIM) $(MADSSIMGEO) -e \
		$(MADSSIM) $(NAME) -$(NAME).cmd -nographics

set-project: $(PROJECT_LIST) $(PROJDIR) set-project.el
	# set current project by changing the index number on the
	# first line of the file $(PROJECT_LIST) to be
	# the position of the current project in the 0-based
	# one-project-per-line list that follows in the file.
	#
	@check-current-project $(PROJECT_LIST) $(PROJDIR)/ || \
	emacs -batch $(PROJECT_LIST) -l $(shell /bin/pwd)/set-project.el \
	-f save-buffer -kill >/dev/null

$(PROJDIR)/$(NAME).cir: $(NAME).cir $(NAME).inc fix.cir.el
	# fix .cir file by running emacs in batch mode
	emacs -batch $(NAME).cir -l fix.cir.el -f save-buffer -kill >/dev/null
	@echo "Edit $(NAME).cir now, if you need to or want to." | \
	alert button="Click here when ready to proceed."
	cp -p $(NAME).cir $@

$(NAME).inc: $(NAME).analog-voltages $(NAME).total-time $(NAME).ini
	echo "* "$@ > $@
	cat $(NAME).analog-voltages >> $@
	cat $(NAME).total-time >> $@
	cat $(NAME).ini >> $@

$(NAME).1 $(NAME).cmd viewsim.ini : $(NAME).ppl
	if [ -r $(NAME).sed ]; then \
	sed -f $(NAME).sed < $(NAME).ppl > $(NAME).ppl.tmp;\
	mv $(NAME).ppl.tmp $(NAME).ppl; \
	fi
	$(SPPLICE) $(NAME) > $(NAME).spplice 2>&1
	sed 's/\.00//g' < viewsim.ini > viewsim.ini.tmp
	mv viewsim.ini.tmp viewsim.ini

$(NAME).dit:
```

```
        @echo -n T
        echo TICKSIZE $(TICKSIZE)ps | \
        sed -e 's/\.00//g' -e 's/000ps/ns/g' > $@
        @echo -n T
        echo STEPSIZE $(STEPSIZE)ps | \
        sed -e 's/\.00//g' -e 's/000ps/ns/g' >> $@

$(NAME).total-time: $(NAME).cmd viewsim.ini
        @echo -n ".PARAM TICKSIZE = " > $@
        grep TICKSIZE viewsim.ini | cut -d" " -f2 >> $@
        @echo -n ".PARAM TOTALTIME = " >> $@
        awk '/SIM/{print $$2}' $(NAME).cmd | \
        sed -e 's/.000ns//' -e 's/.000us/000/' | \
        awk 'BEGIN {total = 0} {total += $$1} END {print total "ns"}' \
        >> $@

$(NAME).analog-voltages:
        # avoltages is a Tcl/Tk script that prompts for Vmin and Vmax
        # (and calculates Vmid = (Vmin+Vmax)/2.0)
#        avoltages > $@
        @echo ".PARAM VMIN = 0.00V  VMID = 2.50V  VMAX = 5.00V" > $@

$(PROJECT_LIST) $(PROJDIR):
        @echo 'Mod1<key>x' > powerview-create-project.xse
        # pdc is the command abbreviation for "project (viewdraw) create"
        @echo "pdc $(PROJDIR)" | xsekey >> powerview-create-project.xse
        toolwait powerview > /dev/null
        xse -window $(PVNAME) -file powerview-create-project.xse
        until [ -d $(PROJDIR)/wir ]; do sleep 1; done
        xse -window $(PVNAME) 'Mod1<Btn3Down>' '<key>Return'

set-project.el:
        @echo "(setq make-backup-files nil)" > $@
        @echo "(goto-char (point-min))" >> $@
        @echo "(if (search-forward \"$(PROJDIR)/\" nil t)" >> $@
        @echo "  (let ((line-number (count-lines (point-min) (point))))" >> $@
        @echo "    (goto-char (point-min))" >> $@
        @echo "    (kill-line)" >> $@
        @echo "    (insert (format \"%d\" (- line-number 2)))))" >> $@

fix.cir.el:
        @echo "(setq make-backup-files nil)" > $@
        @echo "(goto-char (point-min))" >> $@
        @echo "(forward-line 3)" >> $@
        @echo "(open-line 2)" >> $@
        @echo "(setq fname (file-name-nondirectory (buffer-file-name)))" >> $@
```

```
@echo "(let ((file (concat" >> $@
@echo "              (substring fname 0" >> $@
@echo "                (or (string-match \".cir\" fname) 0))" >> $@
@echo "              \".inc\")))" >> $@
@echo "  (if (file-readable-p file)" >> $@
@echo "      (insert-file-contents file)))" >> $@
@echo "" >> $@
@echo "(goto-char (point-min))" >> $@
@echo "(if (search-forward \"   NP6 \" nil t)" >> $@
@echo "    (progn (forward-word 2) (kill-line)" >> $@
@echo "           (insert \" VMIN\")))" >> $@
@echo "" >> $@
@echo "(goto-char (point-min))" >> $@
@echo "(if (search-forward \"   NP7 \" nil t)" >> $@
@echo "    (progn (forward-word 2) (kill-line)" >> $@
@echo "           (insert \" VMID\")))" >> $@
@echo "" >> $@
@echo "(goto-char (point-min))" >> $@
@echo "(if (search-forward \"   NP8 \" nil t)" >> $@
@echo "    (progn (forward-word 2) (kill-line)" >> $@
@echo "           (insert \" VMAX\")))" >> $@
@echo "" >> $@
@echo "(goto-char (point-min))" >> $@
@echo "(if (search-forward \".TRAN \" nil t)" >> $@
@echo "    (let ((file (concat" >> $@
@echo "                  (substring fname 0" >> $@
@echo "                    (or (string-match \".cir\" fname) 0))" >> $@
@echo "                         \".total-time\")))" >> $@
@echo "      (if (file-readable-p file)" >> $@
@echo "          (progn (kill-line)" >> $@
@echo "                 (insert-file-contents file)" >> $@
@echo "                 (zap-to-char 1 ?=)" >> $@
@echo "                 (delete-char 2)" >> $@
@echo "                 (end-of-line)" >> $@
@echo "                 (zap-to-char 1 ?=)" >> $@
@echo "                 (delete-char 1)))))" >> $@
@echo "" >> $@
@echo "(goto-char (point-min))" >> $@
@echo "(if (search-forward \".MODEL D2A\" nil t)" >> $@
@echo "    (if (search-forward \"TIMESTEP=0.1NS\" nil t)" >> $@
@echo "        (progn (zap-to-char -1 ?=) (insert \"TICKSIZE\"))))">>$@
@echo "" >> $@
@echo "(goto-char (point-min))" >> $@
@echo "(if (search-forward \"VLD2A VLD2A 0 DC \" nil t)" >> $@
@echo "    (progn (kill-line) (insert \"VMIN\")))" >> $@
@echo "" >> $@
```

```
        @echo "(goto-char (point-min))" >> $@
        @echo "(if (search-forward \"VHD2A VHD2A 0 DC \" nil t)" >> $@
        @echo "    (progn (kill-line) (insert \"VMAX\")))" >> $@
        @echo "" >> $@
        @echo "(goto-char (point-min))" >> $@
        @echo "(if (search-forward \".MODEL A2D\" nil t)" >> $@
        @echo "    (if (search-forward \"TIMESTEP=0.1NS\" nil t)" >> $@
        @echo "        (progn (zap-to-char -1 ?=) (insert \"TICKSIZE\"))))">>$@
        @echo "" >> $@
        @echo "(goto-char (point-min))" >> $@
        @echo "(if (search-forward \"VREFA2D VREFA2D 0 DC \" nil t)" >> $@
        @echo "    (progn (kill-line) (insert \"VMIN\")))" >> $@

.cleanup:
        @echo "#!/bin/csh -f" > $@
        @echo -n "/bin/rm -f " >> $@
        @echo "$(NAME).{1,cir,cmd,inc,ppg,spplice,tmp,total-time}" >> $@
        @echo "/bin/rm -f $(NAME).analog-voltages" >> $@
        @echo "/bin/rm -f set-project.el fix.cir.el viewsim.ini" >> $@
        @echo "/bin/rm -f spcurins.ins powerview-create-project.xse" >>$@
        @echo "/bin/rm -f .cleanup" >>$@
        @chmod 755 $@
        @echo "adsim is done." | alert \
        button="Type \".cleanup\" to remove all generated files."
```

```
# File: dsim.mk
#
# Makefile to do digital simulation using viewlogic (viewsim)
#
# NAME comes from the dsim script that invokes make with this Makefile
# (e.g., NAME=dtest)
#
PVNAME = 'Powerview Cockpit'
PVDIR = $(shell echo $(WDIR) | cut -d: -f1)
PROJDIR = $(PVDIR)/$(NAME)
PROJECT_LIST = $(PVDIR)/vf/project.lst
SPPLICE = spplice -V -f
VIEWSIM = viewsim

all: .setup goal .cleanup

.setup: $(NAME).ppl $(NAME).src set-project

goal: $(PROJDIR)/$(NAME).wfm
	if [ -r $(PROJDIR)/.streamed ]; then \
	rm $(PROJDIR)/.streamed; else \
	cd $(PROJDIR); viewtrace $(NAME).wfm; fi

$(NAME).ppl:
	@echo "After writing $(NAME).ppl" | alert button="click here"
	@test -f $@ || (echo No such file: $@ && false)

$(NAME).src:
	@echo "After writing $(NAME).src" | alert button="click here"
	@test -f $@ || (echo No such file: $@ && false)

set-project: $(PROJECT_LIST) $(PROJDIR) set-project.el
	# set current project by changing the index number on the
	# first line of the file $(PROJECT_LIST) to be
	# the position of the current project in the 0-based
	# one-project-per-line list that follows in the file.
	#
	@check-current-project $(PROJECT_LIST) $(PROJDIR)/ || \
	emacs -batch $(PROJECT_LIST) -l $(shell /bin/pwd)/set-project.el \
	-f save-buffer -kill >/dev/null

$(PROJDIR)/$(NAME).wfm: $(NAME).1 $(NAME).cmd viewsim.ini
	cp -p $(NAME).1 $(PROJDIR)/wir
	# add digital watch list to $(NAME).cmd
	echo -n "watch " > $(NAME).tmp
	-grep "^wave" $(NAME).cmd | cut -d" " -f3- >> $(NAME).tmp
```

```
        sed '/^wave/r $(NAME).tmp' < $(NAME).cmd |\
        sed 's/QUIT/ECHO Reached end of $(NAME).cmd file: enter q to quit./'\
        > $(NAME)-cmd.tmp; mv $(NAME)-cmd.tmp $(NAME).cmd
        cp -p $(NAME).cmd viewsim.ini $(PROJDIR)
        cd $(PROJDIR); vsm $(NAME) > /dev/null 2>&1;\
        $(VIEWSIM) $(NAME) -$(NAME).cmd > $(PROJDIR)/.streamed

$(NAME).1 $(NAME).cmd viewsim.ini : $(NAME).ppl $(NAME).src $(NAME).dit
        $(SPPLICE) $(NAME) > $(NAME).spplice 2>&1
        sed 's/\.00//g' < viewsim.ini > viewsim.ini.tmp
        mv viewsim.ini.tmp viewsim.ini

$(NAME).dit:
        @echo TICKSIZE 100.00ps  > $@
        @echo STEPSIZE 100.00ns >> $@

$(PROJECT_LIST) $(PROJDIR):
        toolwait powerview > /dev/null
        @echo 'Mod1<key>x' > powerview-create-project.xse
        # pdc is the command abbreviation for "project (viewdraw) create"
        @echo "pdc $(PROJDIR)" | xsekey >> powerview-create-project.xse
        xse -window $(PVNAME) -file powerview-create-project.xse
        until [ -d $(PROJDIR)/wir ]; do sleep 1; done
        xse -window $(PVNAME) 'Mod1<Btn3Down>' '<key>Return'

set-project.el:
        @echo "(setq make-backup-files nil)" > $@
        @echo "(goto-char (point-min))" >> $@
        @echo "(if (search-forward \"$(PROJDIR)/\" nil t)" >> $@
        @echo "  (let ((line-number (count-lines (point-min) (point))))" >> $@
        @echo "    (goto-char (point-min))" >> $@
        @echo "    (kill-line)" >> $@
        @echo "    (insert (format \"%d\" (- line-number 2)))))" >> $@
        @echo "(delete-file \"$(shell /bin/pwd)/$@\")" >> $@

.cleanup:
        @echo "#!/bin/csh -f" > $@
        @echo -n "/bin/rm -f " >> $@
        @echo "$(NAME).{1,cir,cmd,ppg,spplice,tmp,vsm}" >> $@
        @echo "/bin/rm -f set-project.el viewsim.ini spcurins.ins" >> $@
        @echo "/bin/rm -f powerview-create-project.xse" >>$@
        @echo "/bin/rm -f .cleanup" >>$@
        @chmod 755 $@
        @echo "dsim is done." | alert \
        button="Type \".cleanup\" to remove all generated files."
```

REFERENCES

[1] AMBLER, A. L. Generalizing the sheet language paradigm. In *Visual Languages and Applications, Languages and Information Systems Edition.* Plenum Press, New York, 1990, pp. 327–346.

[2] ANDERSON, R. H., AND SHAPIRO, N. Z. Beyond user friendly: Easy to.... *EDUCOM Review 24*, 3 (1989), 50–54.

[3] BALDASSARI, M., BRUNO, G., AND CASTELLA, A. PROTOB: an object-oriented CASE tool for modelling and protyping distributed systems. *Software—Practice and Experience 21*, 8 (1991), 823–844.

[4] BATE, J. S. J., AND VADHIA, D. B. *Fourth-Generation Languages under DOS and UNIX.* BSP Professional Books, Oxford, 1987.

[5] BEERI, C., AND RAMAKRISHNAN, R. On the power of Magic. In *Proceedings of the ACM Symposium on Principles of Database Systems* (Mar. 1987), pp. 269–283.

[6] BOLT, R. A. Conversing with computers. In *Readings in Human-Computer Interaction: A Multidisciplinary Approach.* Morgan-Kaufmann, Los Altos, California, 1987, ch. 14.

[7] BORG, K. Visual programming and UNIX. In *1989 IEEE Workshop on Visual Languages* (1989), pp. 74–79.

[8] BROOKS, F. P. Plan to throw one away. In *The Mythical Man-Month.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1975, ch. 11.

[9] BROOKS, F. P. No silver bullet: Essence and accidents of software engineering. *IEEE Computer 20*, 4 (Apr. 1987), 10–19.

[10] BUCK, J., HA, S., LEE, E. A., AND MESSERSCHMITT, D. G. Ptolemy: A platform for heterogeneous simulation and prototyping. In *Proceedings of the 1991 European Simulation Conference* (1991).

[11] CARROLL, J. M. The adventure of getting to know a computer. *IEEE Computer 15*, 11 (1982), 49–58.

[12] CARTER, T. M., SMITH, K. F., JACOBS, S. R., AND NEFF, R. M. Cell matrix methodologies for integrated circuit design. *INTEGRATION, the VLSI Journal 9*, 1 (Feb. 1990), 81–97.

[13] CHAMBERS, C., UNGAR, D., CHANG, B. W., AND HÖLZLE, U. Parents are shared parts of objects: Inheritance and encapsulation in Self. *Lisp and Symbolic Computation 4* (1991), 207–222.

[14] CHANG, S.-K. Visual languages: A tutorial and survey. *IEEE Software 4*, 1 (1987), 29–39.

[15] CHEW, F. F. Beneath the surface of NewWave office. *HP Professional 4*, 4 (Apr. 1990), 44–53.

[16] COLEMAN, D., ARNOLD, P., BODOFF, S., DOLLIN, C., GILCHRIST, H., HAYES, F., AND JEREMAES, P. *Object-Oriented Development: The Fusion Method.* Prentice Hall, New Jersy, 1994.

[17] COURINGTON, W. The ToolTalk service. Tech. rep., SunSoft, Inc., a Sun Microsystems company, 1991.

[18] CRIMI, C., GUERCIO, A., TORTORA, G., AND M.TUCCI. An intelligent iconic system to generate and to interpret visual languages. In *1989 IEEE Workshop on Visual Languages* (1989), pp. 144–149.

[19] DELMAS, S. XF announcements. comp.lang.tcl USENET article dated 26 Apr 1993.

[20] DIGITAL EQUIPMENT CORPORATION, HEWLETT-PACKARD COMPANY, HYPERDESK CORPORATION, NCR CORPORATION, OBJECT DESIGN, INC., AND SUNSOFT, INC. The common object request broker: Architecture and specification. OMG Document Number 91.12.1, Object Management Group, 1991.

[21] EVANS, J. D., AND KESSLER, R. R. DPOS: A metalanguage and programming environment for parallel processing. *Lisp and Symbolic Computation 5*, 1/2 (1992), 105–125.

[22] FELDMAN, S. I. Make—a program for maintaining computer programs. In *Unix Programmer's Manual—Supplementary Documents I.* USENIX Association, 1986.

[23] FERGUSON, I. A. TouringMachines: Autonomous agents with attitudes. *IEEE Computer 25*, 5 (1992), 51–55.

[24] FIDUK, K. W., KLEINFELDT, S., KOSARCHYN, M., AND PEREZ, E. B. Design methodology management: A CAD Framework Initiative perspective. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* (1990), pp. 278–283.

[25] FOWLER, G. A case for make. *Software—Practice and Experience 20*, S1 (1990), 35–46.

[26] FRANKEL, R. ToolTalk in electronic design automation. A technical white paper, SunSoft, Inc., a Sun Microsystems company, 1991.

[27] FRASER, C. W., AND KRISHNAMURTHY, B. Live text. *Software—Practice and Experience 20*, 8 (1990), 851–858.

[28] GU, J., AND SMITH, K. F. A structured approach for VLSI circuit design. *IEEE Computer 22*, 11 (1989), 9–22.

[29] HALBERT, D. C., AND O'BRIEN, P. D. Using types and inheritance in object-oriented programming. *IEEE Software 4*, 9 (1987), 71–79.

[30] HALME, H., AND HEINÄNEN, J. GNU Emacs as a dynamically extensible programming environment. *Software—Practice and Experience 18*, 10 (1988), 999–1009.

[31] HAMILTON, M. Zero-defect software: the elusive goal. *IEEE Spectrum 23*, 3 (Mar. 1986), 48–53.

[32] HAREL, D. Statecharts: A visual formalism for complex systems. In *Science of Computer Programming*, vol. 8. North Holland, 1987, pp. 231–274.

[33] HAREL, D. On visual formalisms. *Communications of the ACM 31*, 5 (1988), 514–530.

[34] HAREL, D. Biting the silver bullet: Toward a brighter future for system development. *IEEE Computer 25*, 1 (1992), 8–20.

[35] HAREL, D., AND ET AL. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering 16*, 4 (Apr. 1990), 403–414.

[36] HARRISON, D. S., MOORE, P., SPICKELMIER, R. L., AND NEWTON, A. R. Data management and graphics editing in the Berkeley design environment. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (Nov. 1986), pp. 24–27.

[37] HARRISON, D. S., NEWTON, A. R., SPICKELMIER, R. L., AND BARNES, T. J. Electronic CAD frameworks. *Proceedings of the IEEE 78*, 2 (Feb. 1990), 393–417.

[38] HARRISON, W. H. Tool integration technologies through the 90's. *ACM SIGPLAN Notices 27*, 11 (Nov. 1992), 90.

[39] HENDRY, D. G., AND GREEN, T. R. G. CogMap: a visual description language for spreadsheets. *Journal of Visual Languages and Computing 4* (1993), 35–54.

[40] HUTCHINGS, B. L. *A Cell-Based Approach to Hierarchical Inter-Domain VLSI Design.* PhD thesis, University of Utah, 1992.

[41] ISON, R. An experimental Ada programming support environment in the HP CASEdge integration framework. In *Lecture Notes in Computer Science, Software Engineering Environments Edition.* Springer-Verlag, New York, 1989, pp. 179–193.

[42] JONES, C. *Programming Productivity.* McGraw-Hill, New York, 1986.

[43] KERNIGHAN, B. W., AND PIKE, R. *The UNIX Programming Environment.* Prentice-Hall, New Jersey, 1984.

[44] KOJIMA, K., MATSUDA, Y., AND FUTATSUGI, S. LIVE—integrating visual and textual programming paradigms. In *1989 IEEE Workshop on Visual Languages* (1989), pp. 80–85.

[45] LANGSTON, P. S. Unix music tools at Bellcore. *Software—Practice and Experience 20*, S1 (1990), 47–61.

[46] LEE, E. A., GOEI, E., , HEINE, H., HO, W., BHATTACHARYYA, S., BIER, J., AND GUNTVEDT, E. Gabriel: A design environment for programmable DSPs. In *Proceedings of the 26th ACM/IEEE Design Automation Conference* (1989), pp. 141–146.

[47] LEVY, L. S. *Taming the Tiger: Software Engineering and Software Economics.* Springer-Verlag, New York, 1987.

[48] LIBES, D. expect: Curing those uncontrollable fits of interaction. *Proceedings of the USENIX Summer Conference* (June 1990), 11–15.

[49] LICKLIDER, T. R. Ten years of rows and columns. *BYTE 15*, 12 (1989), 324–331.

[50] LIEBERHERR, K. J., AND HOLLAND, I. M. Assuring good style for object-oriented programs. *IEEE Software 6*, 9 (1989), 38–48.

[51] LINOS, P. ToolCASE: a repository of CASE tools. *Software Engineering Notes 17*, 2 (Apr. 1992), 74–78.

[52] LINTON, M. A., CALDER, P. R., INTERRANTE, J. A., TANG, S., AND VLISSIDES, J. M. *Ibuild User's Guide.* Leland Stanford Junior University, 1991.

[53] LINTON, M. A., VLISSIDES, J. M., AND CALDER, P. R. Composing user interfaces with InterViews. *IEEE Computer 22*, 2 (1989), 8–22.

[54] LIPPMAN, S. B. C++ Primer, second ed. Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

[55] MAHLING, A., HERCZEG, J., HERCZEG, M., AND BOCKER, H.-D. Beyond visualization: Knowing and understanding. In *Visualization in Human-Computer Interaction, 7th Interdisciplinary Workshop on Informatics and Psychology* (May 1988), pp. 16–26.

[56] MALINIAK, L. CAD frameworks ride a rough road to success. *Electronic Design 40*, 16 (1992), 36–42.

[57] MALONE, T. W. What makes things fun to learn? a study of intrinsically motivating computer games. Report CIS-7, Xerox PARC, 1980.

[58] MARTIN, J. *Fourth Generation Languages*, vol. 1. Prentice-Hall, New Jersey, 1985.

[59] MARTIN, R. *The Care and Feeding of the* C++ State Map Parser. Clear Communications Inc., Nov. 1991.

[60] MEYER, B. *Eiffel: The Language.* Prentice-Hall, New Jersey, 1992.

[61] MICHEL, S. Interoperability: Not bad, but no piece of cake. *MacWEEK 6*, 28 (1992), 80–92.

[62] MILNER, R. The elements of interaction. *Communications of the ACM 36*, 1 (1993), 78–89.

[63] MORICONI, M., AND HARE, D. F. The PegaSys System: Pictures as formal documentation of large programs. *ACM Transactions on Programming Languages and Systems 8*, 4 (Oct. 1986), 524–546.

[64] MYERS, B. A. Demonstrational interfaces: A step beyond direct manipulation. *IEEE Computer 25*, 8 (1992), 61–73.

[65] MYERS, B. A., GIUSE, D. A., DANNENBERG, R. B., ZANDEN, B. V., KOSBIE, D. S., PERVIN, E., MICKISH, A., AND MARCHAL, P. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Software 23*, 11 (1990), 71–85.

[66] NEFF, R. M. INSTED: An integrated structured tiling editor. Master's thesis, University of Utah, 1986.

[67] NEFF, R. M. *Versatile Interaction Specification of Tools and Agents.* PhD thesis, University of Utah, 1995.

[68] NELSON, B. *Remote Procedure Call.* PhD thesis, Carnegie-Mellon University, 1981.

[69] NEUMANN, G. A Prolog tutorial. *Proceedings of the Conference on Intelligent Simulation Environments* (1986), 163–164.

[70] NEUMANN, G. Wafe 0.91 available. comp.lang.tcl USENET article number 661, 1992.

[71] NORTON, C. D., AND GLINERT, E. P. A visual environment for designing and simulating execution of processor arrays. In *1990 IEEE Workshop on Visual Languages* (1990), pp. 227–232.

[72] OUSTERHOUT, J. Tcl: An embeddable command language. *Proceedings of the USENIX Winter Conference* (Jan. 1990), 133–146.

[73] PATRICK, SR., P. B. CASE integration using ACA services. *Digital Technical Journal 5*, 2 (1993), 84–99.

[74] PETERSON, J. L. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, New Jersey, 1981.

[75] PRESOTTO, D. L., AND RITCHIE, D. M. Interprocess communication in the Ninth Edition Unix System. *Software—Practice and Experience 20*, S1 (1990), 3–17.

[76] RAMAKRISHNAN, R., SESHADRI, P., SRIVASTAVA, D., AND SUDARSHAN, S. *The CORAL User Manual: A Tutorial Introduction to CORAL.* Computer Sciences Department, University of Wisconsin-Madison, 1993.

[77] RAMAKRISHNAN, R., SRIVASTAVA, D., AND SUDARSHAN, S. CORAL: Control, relations and logic. In *Proceedings of the International Conference on Very Large Databases* (1992).

[78] RAYMOND, D. R. Flexible text display with Lector. *IEEE Computer 25*, 8 (1992), 49–60.

[79] RAZOUK, R. R. The use of petri nets for modeling pipelined processors. In *Proceedings of the 25th ACM/IEEE Design Automation Conference* (1988), pp. 548–553.

[80] REISS, S. P. Connecting tools using message passing in the FIELD environment. *IEEE Software 23*, 7 (1990), 57–66.

[81] REISS, S. P. Interacting with the FIELD environment. *Software—Practice and Experience 20*, S1 (1990), 89–115.

[82] RICH, C., AND WATERS, R. C. Automatic programming: Myths and prospects. *IEEE Computer 21*, 8 (Aug. 1988), 40–51.

[83] SCHEIFLER, R. W. *X Protocol Reference Manual.* O'Reilly and Associates, Inc., Sebastopol, California, 1989.

[84] SCHEIFLER, R. W., AND GETTYS, J. The X Window System. *ACM Transactions on Graphics 5*, 2 (Apr. 1986), 79–109.

[85] SCHEIFLER, R. W., WITH J. FLOWERS, J. G., NEWMAN, R., AND ROSENTHAL, D. *X Window System: The Complete Guide to Xlib, X Protocol, ICCCM, XLFD*, second ed. Digital Press, 1990.

[86] SCHINDLER, M. *Computer-Aided Software Design.* John Wiley & Sons, New York, 1990.

[87] SHNEIDERMAN, B. Direct manipulation: a step beyond programming languages. *IEEE Computer 16*, 8 (1983), 57–69.

[88] SHNEIDERMAN, B. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

[89] SHU, N. C. Visual programming languages: A perspective and a dimensional analysis. In *Visual Languages.* Plenum Press, New York, 1986, pp. 11–34.

[90] SHU, N. C. *Visual Programming.* Van Nostrand Reinhold, New York, 1988.

[91] SHU, N. C. Visual programming: Perspectives and approaches. *IBM Systems Journal 28*, 4 (1989), 525–47.

[92] SINHA, A. Client-server computing: Current technology review. *Communications of the ACM 35*, 7 (1992), 77–98.

[93] STAFF. The case for CASE tools. *IEEE Spectrum 27*, 11 (Nov. 1990), 78–81.

[94] STALLMAN, R. Emacs: The extensible, customizable, self-documenting display editor. Tech. Rep. 519, MIT Artificial Intelligence Lab, Aug. 1979.

[95] STALLMAN, R. *GNU Emacs Manual*, fourth, version 17 ed., Feb. 1986.

[96] STERLING, L., AND SHAPIRO, E. *The Art of Prolog.* The MIT Press, Cambridge, Massachusetts, 1986.

[97] STROUSTRUP, B. *The* C++ Programming Language. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[98] UNGAR, D., AND SMITH, R. B. Self: The power of simplicity. *Proceedings of OOPSLA '87, ACM SIGPLAN Notices 22*, 12 (Dec. 1987), 227–242.

[99] VLISSIDES, J. M., TANG, S., AND BRAUER, C. *InterViews Reference Manual.* Leland Stanford Junior University, 1992.

[100] VOSE, G. M., AND WILLIAMS, G. LabVIEW: Laboratory virtual instrument engineering workbench. *BYTE 11*, 9 (1986), 84–92.

[101] WANG, P. *An Introduction to Berkeley UNIX.* Wadsworth Publishing Company, Belmont, California, 1988.

[102] WASSERMAN, A. I. Tool integration in software engineering environments. In *Lecture Notes in Computer Science, Software Engineering Environments Edition.* Springer-Verlag, New York, 1989, pp. 137–149.

[103] WEISERT, C. Macros for defining C++ classes. *ACM SIGPLAN Notices 27*, 11 (Nov. 1992), 67–76.

[104] WILLIAMS, C. S., AND RASURE, J. R. A visual language for image processing. In *1990 IEEE Workshop on Visual Languages* (1990), pp. 86–91.