A Visual Scripting Language

by

Ernest Alan Idler
B.Sc., University of British Columbia, 1981

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. Hausi A. Müller

[redacted]

Dr. Daniel M. Hoffman

[redacted]

Dr. Kin F. Li

[redacted]

Dr. Warren D. Little

University of Victoria

Supervisor: Professor Hausi A. Müller

ABSTRACT

The Rigi Editor is a practical software development environment for programming-in-the-large based on the Rigi Model. The objective of this thesis is to present Survey, a scripting language, which permits tasks performed through the Rigi Editor interface to be programmed as command scripts. To promote consistency throughout the Rigi Editor interface, the Survey programming environment is iconic, using the same hypertext editing paradigms to maintain its scripts.

UNIX programmers use the find command to perform basic file maintenance. Findtool is an application of the Rigi Editor interface as a visual interface to the UNIX find command. Findtool was implemented as a prototype of Survey to investigate the suitability of the Rigi Editor interface as a visual programming environment.

In the implementation of Findtool and Survey, it was prudent to reuse as much of the previous Rigi development effort as possible. Therefore, these projects provide excellent test cases for current techniques for reusing software. A reusable system design is developed.

Examiners:

[redacted]

Dr. Hausi A. Müller

[redacted]

Dr. Daniel M. Hoffman

[redacted]

Dr. Kin F. Li

[redacted]

Dr. Warren D. Little

Table of contents

List of tables

List of figures

Acknowledgements

Having completed this journey, I know that I could never have come through on my own. There are so many to thank, starting with my thesis committee: Hausi Müller, Dan Hoffman, and Kin Li were each so knowledgeable and always willing to hear how I was progressing. The inspiration for Survey originated with Dr. Müller and thus he was most eager for the slightest bit of news. It was Hausi's patience in reading several preliminary versions that enabled us to strain out these few gems.

My work in the Department of Computer Science gave me the opportunity to pursue graduate studies even though at times people had a terrific time deciding whether I was a student or staff (myself included). My close association with the faculty here in Victoria convinced me that I had academic aspirations.

Many thanks to Karl Klashinsky and Jim Uhl who were the system programmers for Rigi. Without their cooperation (and their code!) I could not have developed the software I did. And thanks to all my fellow graduate students who patiently listened to me fumble through various seminars while this work came together.

Leslie, I do hope you know that I love you even if I neglected you for the past two years. I thank you for selflessly sustaining me in an obsession that I know you will never fully comprehend. And now maybe Aaron, Sarah, and the baby to be named later can have their Daddy back again.

1. Introduction

1.1 Programming environments

As long as there have been computers there has been the accompanying challenge of programming them. One can argue endlessly over the merits of various programming languages, but the fact remains that programming is a creative task which is also intellectually rigorous. A programming language allows a programmer to express the concepts inherent in an application while removing the mundane details of the machine architecture.

To further assist programmers in developing programs, *programming environments* have been implemented to support programmers in their daily routines, providing essential services as requested and further enhancing the programmer's talents. Today, programmers expect to be supplied with editors and debuggers, in addition to their language compiler. Many programming environments possess some of the qualities of expert systems, anticipating the needs of programmers or alerting them to potential errors.

In a *procedural* programming environment, the programmer must specify each step of the computation explicitly [GlTa 84]. Most common programming languages, such as Pascal, are procedural. A programming environment is *non-procedural* if a programmer states only the requirements for the program, that is, sets some constraints on the inputs and outputs, and general specifications on the calculations performed. It

is then the responsibility of the programming environment to produce a program to satisfy those requirements. Statistical modelling packages such as SPSS have been designed non-procedurally [SPSS 83].

Within the class of procedural programming environments there are algorithmic and demonstrational programming environments [GlTa 84]. Computer scientists are most familiar with *algorithmic* programming environments because the conventional wisdom in the field is to design a program based on an algorithmic solution which solves the problem abstractly (without computing any values during the design phase). Since professional programmers often compute some trial values to gain an understanding of a problem before designing an algorithm, demonstrational programming environments have received some research interest. A *demonstrational* programming environment encourages the programmer to demonstrate how to perform different variations of the calculation as it "learns" enough about the problem to develop a program to solve it in the general case.

The programming environments discussed so far are textual in format. With the recent availability of low-cost graphic interfaces, computer scientists have proposed that there may be alternative methods of programming which would make programming easier. *Visual programming environments* employ graphics, supported by text as necessary. The visual programming environments developed to date have mostly been research projects designed to demonstrate what might be done with more effort in the field. No one can tell whether this style of programming will eventually become more

popular than our current programming environments based on text files.

1.2 Software as a reusable resource

> *The main activity of programming is not the origination of new independent programs, but in the integration, modification, and explanation of existing ones.*
>
> *T. Winograd [Wino 84]*

A major software development project involves hundreds of programmers and support personnel. The challenges of managing projects of this magnitude have reached a critical point. Because software development demands such a huge investment, people are applying lessons learned through other engineering disciplines towards solving the software crisis.

Many new electronic products are constructed from basic sub-components borrowed from other suppliers or previous projects. Others are close revisions to familiar technology. *Software reusability* concerns the promotion of software as a reusable resource and the development of techniques and tools to assist software engineers in meeting this objective.

One of the most effective techniques of software reusability is to design popular programs which are widely distributed or run repeatedly. While this is excellent economic justification for developing software in general, this thesis is concerned with the specific challenges of creating new software by reusing existing software.

Reusing others' work benefits developers and users alike in permitting higher-quality products to be produced at lower cost [Wegn 84]. Reuse allows designers,

programmers, and integrators to be more productive by transfering to them the expertise of their collegues through the software reused. It amplifies the skills of good software engineers, allowing more of their talents to be devoted to the problem at hand.

Increased productivity, reliability, and maintainability is dependent on reusing software of high quality. Better software requires greater design, development, and testing effort, and, since programmers' time is money, management must view software as a long term investment to realize the returns of software reusability.

1.3 Approach

The objective of this thesis is to present *Survey*, a *scripting language* for programming the Rigi Editor interface. A scripting language is essential to support software development and project management in a software development environment such as Rigi. A scripting language stores interactive commands for later execution when operations are too laborious or repetitive to be performed manually.

Because such a priority is placed on uniformity in user interfaces [SUN 88b], we have chosen to experiment with Rigi Editor as a visual programming environment for Survey.

Since Survey exists inside the framework of the Rigi Editor, many of the technical decisions cannot be easily changed. Rigi is a software development environment based on SUN Workstations, running the UNIX operating system. Rigi uses primitives from the SunView window system extensively. Since C is the

programming language of choice under UNIX and the only one with direct access to the SunView libraries, and because many C source modules were already developed for the Rigi Editor, all modules for this project are written in C.

The UNIX *find* command provides a simple scripting language to support file system maintenance. Because of the similar structure of the UNIX file system and the Rigi Model, Survey has been designed as a "relative" of *find* to provide the flexibility of command scripts as extensions to the Rigi Editor.

To gain experience with the Rigi Editor interface as a visual programming environment, an intermediate tool was developed. Findtool provides the full functionality of *find* with the visual interface of the Rigi Editor. Findtool is an independent entity in its own right: the visual interface is an improvement over the standard UNIX command line.

Findtool is an amalgam of the Rigi Editor user interface and *find*. Survey naturally evolves from Findtool. It would be desirable to reuse as much as possible of the development effort from each of these projects to minimize design and development time. The reuse of the design of Findtool in Survey contributes to software reusability.

These challenges are essentially an exercise in software reusability. Findtool and Survey provide good test cases for experimenting with trends in software engineering pertaining to software reuse.

1.4 Thesis overview

Chapter 2 of this thesis introduces background material on the following research

topics: visual programming environments, the Rigi software development environment, and software reusability. Chapter 3 discusses a visual interface for *find*. Chapter 4 presents the Survey scripting language and its integration into the Rigi Editor. The services provided within Survey are justified.

The reusable system design developed for Findtool and Survey is presented in Chapter 5. Because Findtool is developed as a prototype of Survey, these software developments constitute a case study in software reusability. A system design to foster software reuse is developed.

Chapter 6 is the thesis epilogue. The major results of this thesis are summarized: evaluations of Findtool and the Survey scripting language; and, a critique of the software reusability techniques employed. Directions for future research are also presented.

2. Related research

2.1 Visual programming environments

In a *visual programming environment* programs are created by designing and positioning graphic objects representing conceptual entities. Graphics must be an integral, not merely decorative, part of a programmer's interaction with the environment. A visual programming environment is termed *iconic* if programming involves selecting and/or composing icons and placing them in proper juxtaposition on the screen [GlTa 84]. *Icons* are images carrying syntactic and semantic interpretations in their programming environment.

Visual programming environments are receiving growing research interest because it is generally believed that interactive graphics may be more intuitive than textual interfaces. According to Tanimoto and Glinert, a well-designed visual programming environment affords many advantages over traditional text-based efforts [GlTa 84, TaGl 86]:

1. Graphics are "friendlier" and can be easier to learn.

2. Graphics approximates a universal language. Other programming languages are typically English-based, creating a language barrier for people from other cultures.

3. Graphics provide a good summary of information. People prefer to view data as a chart rather than as numeric tables. The structure of a problem, or the program which solves it, may be described better graphically. Flowcharts continue to be

demanded in documentation to satisfy people's need to visualize.

4. Researchers have discovered the importance of developing a consistent interface standard for all programs on their systems.

5. Even if visual programming environments never progress beyond the rudimentary level, they can be used as teaching aids for new programmers, easing them into traditional languages as they develop maturity in problem solving. Children and other people who cannot read can still program visually.

Some visual programming environments will be more successful than others. According to Glinert and Tanimoto, the following elements are essential to producing an effective visual programming environment [TaGl 86]:

1. The system platform (hardware and software) must be capable of interacting responsively with the programmer;

2. The visual language must be designed to communicate appropriate concepts while enhancing creativity;

3. The environment must be founded on an effective computational metaphor.

Quick system response is always desirable and interactive graphics compound the volume of screen updating. Computer engineers are continuing to develop faster graphic displays and workstations. New standards will improve the interactive response of graphic software. This is inevitable. It is not that long ago that what we do routinely with micro-computers was considered science fiction.

Graphic images and icons form a large part of a visual programming language.

Programmers use them to define objects in a program. It can be quite a challenge creating icons of appropriate detail, unambiguous in conveying syntactic and semantic meanings. Glinert and Tanimoto suggest consulting a professional graphic designer.

Display space is such a premium that the layout of objects can become unmanageable. With panning and zooming, the programmer controls the level of detail displayed. Scrolling and screen refreshing can be minimized with multiple displays, but keeping the programmer and the system's attention focused on the same monitor is difficult. Over time, the physical strain of turning from one display to another becomes uncomfortable and can accentuate health problems.

Tanimoto and Glinert continually state that clear, consistent metaphor for the act of programming is by far and away the most significant factor in designing an effective visual programming environment. Metaphors are used in great literature because an author can more effectively communicate abstract ideas by appealing to the reader's personal experiences. Likewise, an effective programming metaphor expresses the concepts well that appear in computer programming.

The metaphor employed depends on the style of programming environment the designer envisions. For instance, one intuitive concept is following a path: people learn to crawl early in childhood. However, this metaphor can be expressed in several variations. If the designer wants an algorithmic programming environment to appeal to programmers with some experience, the path could be constructed as a standard flowchart. The path could be traced by an animated character as a demonstrational exercise. Alternatively, rather than focussing on the flow of control, a non-procedural

programming environment could be based on dataflow diagrams.

Pict/D by Glinert and Tanimoto is one implementation of a visual programming environment based on a flowcharting metaphor [GlTa 84]. Pong and Ng achieved the same result using Nassi-Shneiderman Diagrams [NaSh 73] in PIGS [PoNg 83]. Reiss' PECAN system is an extensive effort to expand on these metaphors allowing programmers to generate Pascal programs by constructing flowcharts, N-S diagrams, dataflow diagrams, or finite state automata [Reis 85].

Graphic interfaces have allowed designers the freedom to devise elaborate visual metaphors so imaginative that they have influenced the psychology of programming. *Programming by Rehearsal* is one of these all encompassing visual programming environments using a theatrical motif [FiGo 84]. The program becomes a "production" and the programmer its "director." In this demonstrational system, "performers" are given "cues" to take a position on the "stage," and so on. The system may also be programmed algorithmically in Smalltalk. Finzer and Gould report that students become as comfortable Programming by Rehearsal in just two or three days as they do when learning to program with a conventional programming environment during a one semester course. Students do not even realize that they are programming!

While this is by no means an exhaustive survey, a few more developments are worth mentioning. In Query by Example by Zloof, office workers formulate database queries by designing the bureaucratic forms they work with day after day [Zloo 80]. Glinert has released another visual programming environment, BLOX, that structures

blurbs of Pascal code like pieces in a jigsaw puzzle [Glin 86]. Instead of using keywords as connectives, each control structure has an iconic representation with prongs indicating what other components are necessary to complete the program.

The value of graphic interfaces is subjective and a matter of personal opinion. Because this field is still in its infancy, researchers are still experimenting with visual programming metaphors and interface techniques in an effort to discover what will be most successful. The visual programming environments produced to date have little practical application because they have not been designed to compete with traditional programming environments.

2.2 The Rigi project

A programming environment supports a programmer in developing a single *module*, i.e., compilation unit. The integration of the hundreds or even thousands of modules comprising a typical software project is a tremendous challenge. *Software development environments* assist in managing large software projects through all phases of their life cycles.

The objective of the K2 project [MHHL 87], underway at the University of Victoria, is to advance research in software development environments. The software development environment is specified by the Rigi[1] Model as a *semantic network data model* [Müll 86]. System components constitute nodes in a network or graph and connections between nodes imply relationships between the system components. A

[1] Rigi is a mountain in Switzerland.

conceptual view of the Rigi Model is depicted in Figure 2.1.

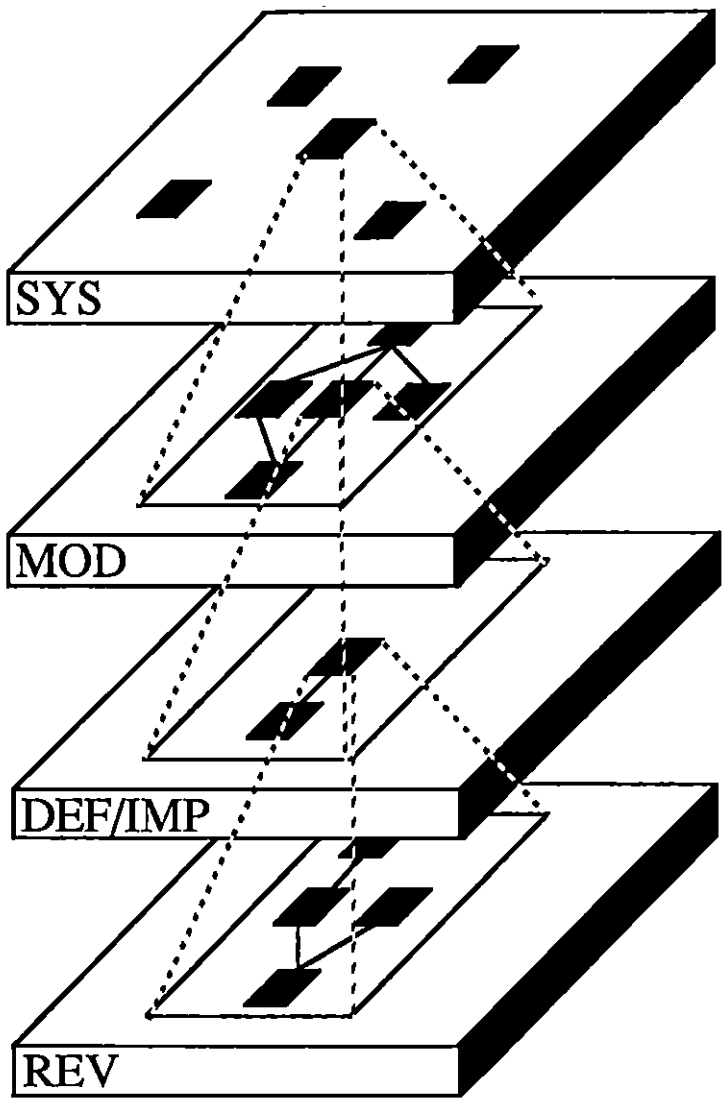


Figure 2.1: The Rigi Model [MüKl 88]

The Rigi Model employs abstraction mechanisms of classification, aggregation, generalization, and grouping as *organizational axes* to structure the knowledge

represented in its underlying semantic network [BrMS 84]. Another organizational axis is achieved by applying an abstraction recursively, generating an hierarchy.

The eleven object classes and the three dependency classes defined by the Rigi Model are listed in Tables 2.1 and 2.2, respectively.

sys	software systems and subsystems
mod	software modules
def	module specifications
imp	module implementations
gen	specification variants (*generics*)
alt	alternative implementations
pro	programs (modules with no specifications)
rev	source revision
dat	data (e.g., for testing)
doc	documentation
pic	documentation (figures)

Table 2.1: Rigi object classes

structure	component A requires resources from B
change	a change in A requires B to be re-compiled (as well)
semantic	hidden dependency a designer wants to document

Table 2.2: Rigi dependency classes

Aggregation occurs when more complex components are constructed from atomic components (i.e., *rev*, *pic*, *doc*, and *dat*). Objects may only be composed within a given layer. The four layers in the *aggregation hierarchy* are illustrated in Figure 2.1. Table 2.3 cross-references the objects belonging to each hierarchy layer. *Accessory* objects (*pic*, *dat*, and *doc*) can appear at any level, since they do not contribute to the system structure.

The *gen* object incorporates generalization into the Rigi Model. The converse of generalization is *specialization.*

system layer	sys
module layer	pro, mod, sys
definition/implementation layer	def, imp, gen, alt
revision layer	rev

Table 2.3: Membership of objects within the Rigi aggregation hierarchy

The Rigi Editor is an implementation of the Rigi Model [Müll 86, MüKl 88]. A full description of the Rigi Editor is found in Klashinsky's M.Sc. thesis [Klas 88]. Figure 2.2 shows a snapshot of a Rigi Editor session in progress.

Hypertext systems, as proposed by Nelson, are ideally suited to displaying and interacting with a database of networked nodes [Nels 80]. Since the Rigi Model is a semantic network, it is natural for the Rigi Editor to be implemented as a hypertext

system. The Rigi Editor supports the abstraction mechanisms defined by the Rigi Model and enforces its semantics, in addition to the hypertext interface.

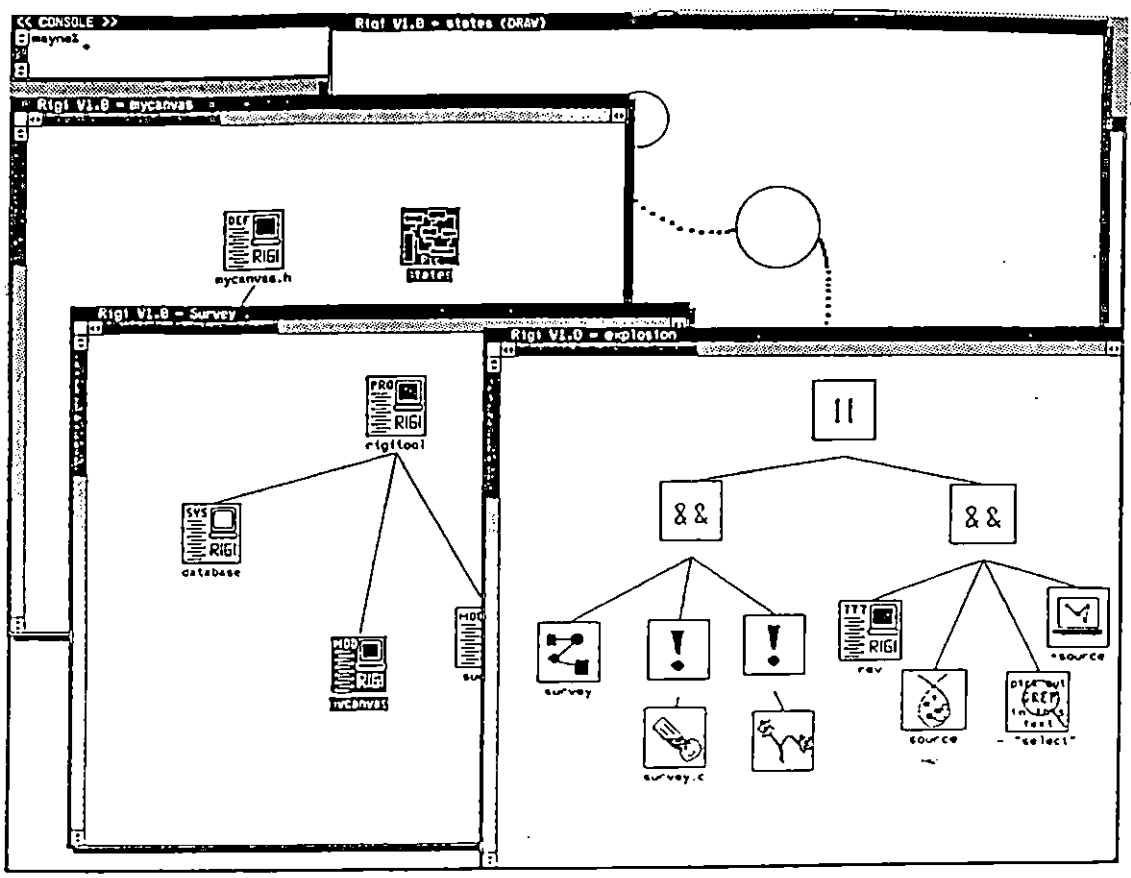


Figure 2.2: Rigi Editor session

A brief discussion of how the abstraction mechanisms of the Rigi Model are implemented in the Rigi Editor is pertinent to the design of Survey. The semantic networks comprising the Rigi Model are shown in windows as graphs of icons.

Dependencies are drawn as arcs, from the bottom of the source object to the head of its dependant; different types of dependencies have arcs of varying patterns. Each object class has its own distinctive icon, including the name of each individual object. The aggregation hierarchy is traversed by opening icons into windows or "iconifying" windows into icons. Grouping is controlled by placing related objects in close proximity to each other or by marking a group of icons as the *selected set.*

2.3 Software reusability

2.3.1 Challenges in reusing software

> *The reuse of program code alone has almost no value .*
>
> *P. Freeman [Free 83].*

Software reusability is no accident. Like anything of value, software reuse does not occur without effort. First of all, software engineers must recognize that reused software contributes to overall software quality and lowers its cost. Then, they must commit to making reusable software available and profitable.

Even if a software engineer employs software reusability as a part of professional practice, there are severe obstacles to being successful at it:

1. Searching for components sufficiently similar to reuse;

2. Understanding enough about a component to make an informed decision;

3. Modifying components and composing new ones.

An expert system could present a few of the more likely candidates from a

database. Responsible software engineers would not reuse a software component if they were unsure of its reliability or performance. There must be assurance that formal release procedures have placed quality controls on the software. Documentation standards permit an accurate assessment of the software to be made quickly.

Many programmers consider it a great accomplishment if they can call a subroutine or copy some source code. If software is to be reused on a "grander scale," programmers need to incorporate their colleagues' designs at a much higher level. While programmers make implicit references to the analysis required to produce code that they reuse, Freeman proposes that software reusability needs to look beyond mere code reuse to design reuse, specification reuse, and application model reuse [Free 83].

2.3.2 Techniques for software reusability

2.3.2.1 Abstraction

> *Information hiding and abstraction are two sides of the same coin. The value of abstraction has always been that results developed in terms of an abstraction may be reused for any valid model of that abstraction. We have shown in this paper that by developing and catologing abstractions, we can greatly increase the likelihood that some of the software we write will be reused.*
>
> *D. L. Parnas, P. C. Clements, D. M.Weiss [PaCW 83]*

The key element in software design applied in these reusability techniques is abstraction. *Abstraction* simplifies a problem, emphasizing some characteristics while suppressing others. Abstraction is popular in computer science because it manages the intellectual complexity of programs [Shaw 84]. Abstraction yields simpler, better

understood, modules which are excellent candidates for reuse.

The first attempts at abstraction were concerned with procedural abstraction to provide structured programming. Abstraction has grown to imply a separation between the specification of a program and its implementation. An *abstract data type* requires a programmer to use an interface of prescribed access procedures to manipulate internal data structures. The *encapsulation* of data structures within an abstract data type leads to more secure modules because it is easier to detect errors if access is controlled by the interface. Parnas expresses the encapsulating effects of abstraction with the term *information hiding* [Parn 72]. Because abstract data types are so attractive in software design, programming languages providing encapsulation rules have become quite popular: Modula-2 [Wirt 83] and Ada [Ichb 79] both package private data structures within a module.

Since abstract data types are simple and reliable they are precisely what are required for effective software reuse. It is relatively easy to place them as building blocks in larger systems. If Parnas' advice is taken into account, modules are designed as abstract data types to encapsulate data structures and algorithms that are most likely to change in response to changes in specifications [Parn 72]. Therefore, by planning for change, reasonable modifications can be accommodated. Abstraction creates the building blocks to be re-assembled; the shape of the blocks, the designer anticipates reusing, dictates the abstraction techniques applied.

Still, as Parnas, Clements, and Weiss acknowledge, there remains a huge void in

placing the associated design information into the hands of the programmers who need to reuse it [PaCW 83]. Cedar provides a *module interconnection language* as part of its software development environment to describe how abstract data types inter-relate [Dona 85].

2.3.2.2 Code libraries

Software reusability has traditionally involved placing object code in user accessible libraries to be linked as subroutines [Wegn 84]. Libraries are most successful in applications where standards are well defined and are not likely to change, such as mathematics and string processing.

Smaller subroutines tend to be more reusable than larger ones, since they tend to be simple and context-free. However, the amount of code reused by each call shrinks.

Libraries tend to be rather inflexible. Written in a particular programming language, they impose its data types and subroutine linkage conventions. It is difficult to specify optional parameters or refine algorithms implemented in the library.

Generic or *polymorphic* typing schemes attempt to make code libraries more universal. Strongly-typed programming languages detect simple coding errors during compilation. However, the compiler does not permit a general-purpose subroutine library to perform simple functions, such as queuing of objects of an unspecified type. This incompatibility must be circumventable, since every imperative programming language has at least one example of a polymorphic function: the assignment operator.

Two obvious solutions are: (1) neglecting to enforce *separate compilation*,

reverting to *independent compilation* which does not require consistent types across module interfaces; or, (2) abandoning data types altogether, declaring an universal type. Neither of these alternatives is desirable because they defeat the security of the typing scheme.

Polymorphism takes many forms: Ada has generic procedures, a context-sensitive macro expansion creating as many versions of a procedure as are necessary; Russell allows types as parameters [BoDD 85] and also implements *type inference*, where the compiler determines the type of objects from their context.

Despite their advantages, polymorphic programming languages are not widely used because of their overheads. On the one hand, the syntactic view of polymorphism exhibited in Ada can generate large bodies of almost identical code and does not permit recursive routines; on the other hand, semantic interpretations require extensive run-time support to perform type checking during execution.

Promoting functions to first-class objects in some programming languages allows programmers to install call-back routines in libraries. Code skeletons are more flexible because the algorithms within the library can be customized by the caller. However, as the skeleton becomes more "boney," more preparation is required before it can be reused.

2.3.2.3 Inheritance

Programmers select software for reuse because they are confident that it meets

their requirements or can easily be modified to do so. Often something similar to, but slightly different from, software already available is required.

Object-oriented programming languages provide a notational vehicle for expressing such relationships in a hierarchy of abstract data types (*classes*). Each sub-class *inherits*, implicitly, the data structures and operations of its *super-class*. A class may augment or replace inherited attributes with its own. Some object-oriented languages provide for private attributes not accessible to sub-classes.

The first object-oriented programming language featuring classes was SIMULA 67 [DMNy 70], later Smalltalk was developed [GoRo 83], and a more recent example is C++ [Stro 86]. C++ is designed to enhance the reusability of its parent, C.

Inheritance has the benefit of including structural and design information in the library. However, a hierarchical ordering is not suitable for expressing some natural relationships such as mutually recursive definitions.

2.3.2.4 Software generators

> *The change from a program-centred to a data-centred view of programming is comparable to the shift from the earth-centred to the sun-centred view of the solar system brought about by the Copernican revolution.*
>
> *C. Bachman [Bach 73]*

When programmers say they are programming, they usually imply creating source code. Suppose a program exists to create source code for certain specifications for a given application domain. Then programming could take another focus: programs

are produced by generating new specifications as data for this program.

Code generation is also called *meta-programming*, since many applications exploit regular patterns in their structure. The format of the data can be obscure, but knowledge acquisition systems are being developed to assist users. Code generation affords independence from a particular programming language. Algorithms employed by code generators can evolve to bring efficiency improvements, without changes to the data.

The most serious deficiency of code generators is that the output from one generator may be incompatible with the next. A code generator may produce huge amounts of inefficient code. While source code is the result, it may not be very readable by humans: is it better to perform maintenance on the resultant code or re-run the generator?

2.3.2.5 Extensible applications

Early in the development of programming languages computer scientists experimented with *extensible languages* which allowed programmers to devise their own language features (i.e., control structures as well as data structures). These languages proved to be unworkable because of the difficulty in defining interesting extensions and keeping independent variations consistent [Shaw 84]. Nevertheless, software engineers can apply these lessons today to improve software reusability.

System developers could explicitly code every function users may require. However, for any application there is a minimal set of primitive operations. A

scripting language can compose more complex functions from these primitives. These *scripts* may be kept in-house or the interpreter can be distributed so users can create their own.

Command scripts have become very popular. UNIX features several scripting languages: *sh*, *csh*, *sed*, *awk* and *find* [SUN 88a, Kern 84]. Command scripts have the advantage of being extremely flexible; the application can evolve in ways never envisioned when it was first developed. Users appreciate being able to craft their own functions and store tedious, time-consuming functions as scripts until required. Efficiency improvements can be realized through upgrading the algorithms employed within the scripting language without damaging functional scripts.

2.4 Summary

This chapter reviewed some of the research related to this thesis: visual programming environments and software reusability. The Rigi Model was introduced primarily to familiarize readers with its terminology.

3. Findtool: An iconic interface for the find command

This chapter presents Findtool, a visual interface for the UNIX *find* command. It is assumed that the reader is familiar with the *find* command syntax and semantics as described in the SUN OS Commands Reference Manual [SUN 88a]. The reader is assumed to be comfortable with the operation of suntools, the SUN windowing environment, in general, and the Rigi Editor, in particular.

3.1 Introduction

Findtool is a prototype of the Survey scripting language to assess the Rigi Editor interface as a visual programming environment. Findtool is an excellent choice as a prototype because both *find* and the Rigi Editor operate on hierarchies of objects.

The design of Findtool is faithful to the original *find* command. Findtool has the full functionality of *find*; only the interface is improved. Every attempt is made to follow the style and concepts introduced by the Rigi Editor. Since the Rigi Editor is designed in accordance with the SUN User Interface Conventions [SUN 88b], a regular user of suntools should have little difficulty becoming familiar with the Findtool Editor.

With *find*, textual predicates and boolean operators are assembled into an expression. The structure of the expression is controlled by parentheses and an implicit

left-to-right scanning of the expression, as in mathematics. Each predicate has a keyword.

An expression defines an *abstract syntax tree.* It can be readily assumed that Findtool users are comfortable with trees as an equivalent representation of the command-line notation. A two-dimensional drawing "canvas" affords greater latitude for expressing relationships. Why not describe the abstract syntax tree graphically?

Keywords are replaced by bit-mapped icons depicting the intent of the predicate and operators. The structure of the abstract syntax tree determines the evaluation precedence. In the case of predicates with more than one sub-expression, a left-to-right rule is still necessary to resolve ambiguities. While re-designing the interface the `-a` (and) and `-o` (or) operators were extended to accept multiple arguments, making the abstract syntax tree easier to manipulate. (Left-to-right short-circuit evaluation of the expression, as in C or Modula-2, still applies.) *Find* already has some support for this because a missing conjunctive is assumed to be a `-a`.

3.2 Findtool Editor

The Findtool Editor is a tool for displaying, editing, and executing Findtool expressions. A Findtool session in progress is depicted in Figure 3.1. The tool displays an iconic expression in a SunView drawing canvas. The smaller rectangular window in the foreground is a standard SunView pop-up edit window for entering parameters for some predicates. In the background is the Findtool Output Window to

display any output from the Findtool script.

A prospective user of Findtool should become familiar with the Rigi Editor. Firstly, the Rigi Editor serves as an example of a standard SunView *tool* (application). And secondly, the Rigi Editor incorporates interface standards from the Apple Macintosh which may not be familiar to other suntools users.

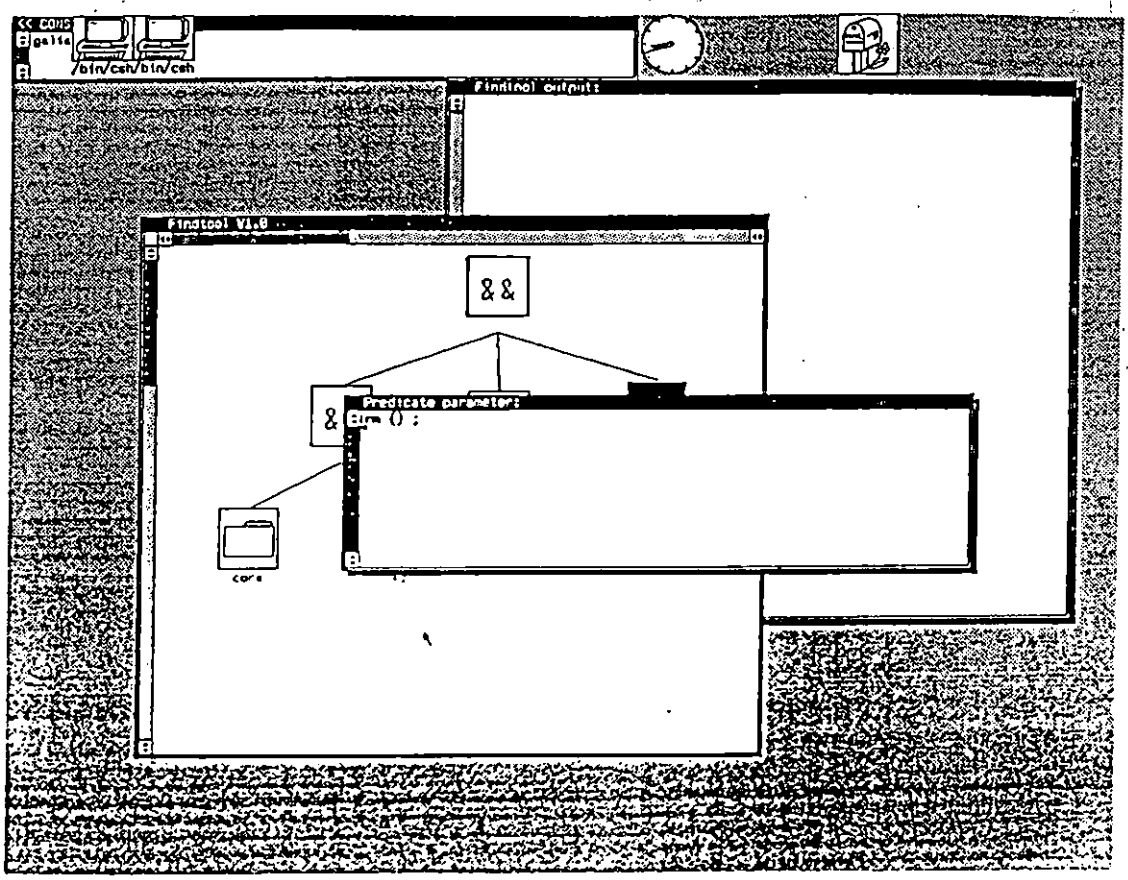


Figure 3.1: Findtool Editor session

Like the Rigi Editor, the Findtool Editor manipulates directed graphs displayed on a SunView drawing canvas. Whereas in the Rigi Editor the figures displayed represent software components and their inter-dependencies, the Findtool Editor displays and

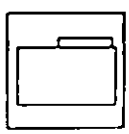
manipulates Findtool expressions as abstract syntax trees. Even so, the editing paradigms remain the same. The operation of Findtool is detailed in Appendix B.

3.3 Findtool predicates

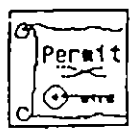
Findtool supports all the *find* predicates described in Appendix A. However, there are two departures from the *find* predicates:

1. The `-o` operator permits more than two operands.

2. A bug in the `-newer` predicate is corrected. *Find* stores the modification time of the file parameter in a global variable during parsing of the *find* expression, which is fine as long as no more than one `-newer` predicate is contained in a *find* command. Findtool queries the modification time-stamp for each file it processes.



-name filename



-perm onum

-fstype type

-type c

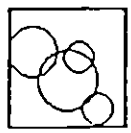
-user uname

-nouser

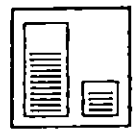
-group gname

-nogroup

-newer file

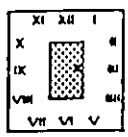


-links n

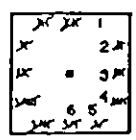


-size n

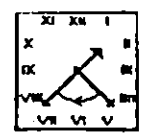
-inum n



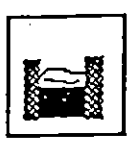
-atime n



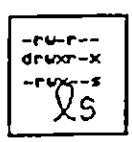
-mtime n



-ctime n



-print

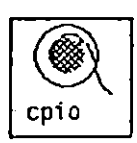


-ls

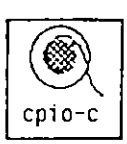
-prune

-xdev

-depth



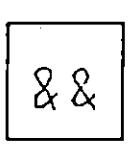
-cpio device



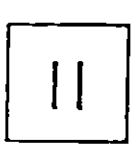
-ncpio device

-exec command

-ok command



expr **-a** expr



expr **-o** expr

! expr

3.4 Findtool example: rgrep

Grep is a standard UNIX utility for scanning files for search patterns. All lines

in the file containing a string matching the pattern are placed on standard output. For example, the following command searches for all uses of the function `printf` in all C source files in the current directory

```
grep printf *.c
```

One deficiency of *grep* is that it lacks a recursive option to automatically search sub-directories. Since this is precisely the purpose of *find*, many UNIX users quickly write this little *find* command script to search file system hierarchies

```
find . -name '*.c' -exec grep printf {} \;
```

How would this *find* command script look as a Findtool expression? An equivalent Findtool expression could be constructed as follows:

1. Findtool is started specifying the pathname(s) to search;

2. The three Findtool predicates, *-name*, *-exec*, and *-a*, and their parameters are inserted as described in Appendix B to form the expression in Figure 3.2;

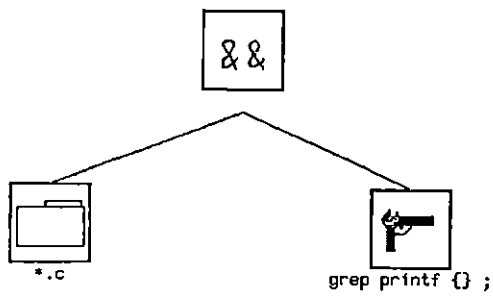


Figure 3.2: rgrep Findtool expression

3. The Findtool expression is executed and any lines found by *grep* are appended to the Findtool Output Window.

3.5 Maketool: using Findtool to support re-compilation

Make is an effective tool for expressing re-compilation dependencies [Feld 79, SUN 88a]. Simulating a modest *makefile* should exercise Findtool enough to expose most of its strengths and weaknesses as a visual programming environment.

3.5.1 The re-compilation problem

A software system may consist of hundreds of inter-dependent modules. If one module is dependent on objects declared in another module, the first module *imports* objects *exported* from the second. To incorporate changes to some source files it is necessary to rebuild the project. Not only does the file changed require re-compilation, but all modules importing objects from that module, and then all modules importing objects from those, and so on. The simplest method for determining whether a module requires re-compilation is to compare the time-stamps of the source file with the last version of the project or object file depending on it.

Make is a standard UNIX facility for expressing file dependencies. *Makefiles* function on the basis of targets and dependencies. When a *makefile* is executed for a given target, if a file depending on that target has been modified since the target was last modified, then the listed *shell* commands are executed. Each dependency may be a target with dependencies. A target is reachable by specifying its name explicitly or

through the dependency lists. Cyclic dependencies are not permitted.

Suppose a simple project consists of these simple source files.

hello.h

```
#define HELLO     "Hi.  Everybody!"
```

hello.c

```
#include <stdio.h>
#include "hello.h"
main ()
{
printf ("%s\n", HELLO);
}
```

Let us suppose that the executable for this little project is called `hello`. Then a suitable *makefile* would be *Makefile* 1, below.

Makefile 1

```
CFLAGS = -g

hello: hello.o
      cc $(CFLAGS) -o hello hello.o

hello.o: hello.c hello.h
      cc $(CFLAGS) -c hello.c
```

3.5.2 Applying Findtool to re-compilation

The key predicate is `-newer` to compare time-stamps of files. Re-compilation is performed via the `-exec` predicate. The entire organization of the "*makefile*" will

change to accommodate *find*; even so, *makefiles* may not be fully converted to Findtool scripts. Let us look at a couple of approaches to converting the simple *makefile* from the `hello` project in the previous section to a Findtool script.

First of all, let us try keying on the files in the dependency list. A snapshot of this Findtool script is shown in Figure 3.3. When a file in the dependency list is found, its time-stamp is compared to the time-stamp of the target. If it is `-newer` than the target, then re-compilation is performed.

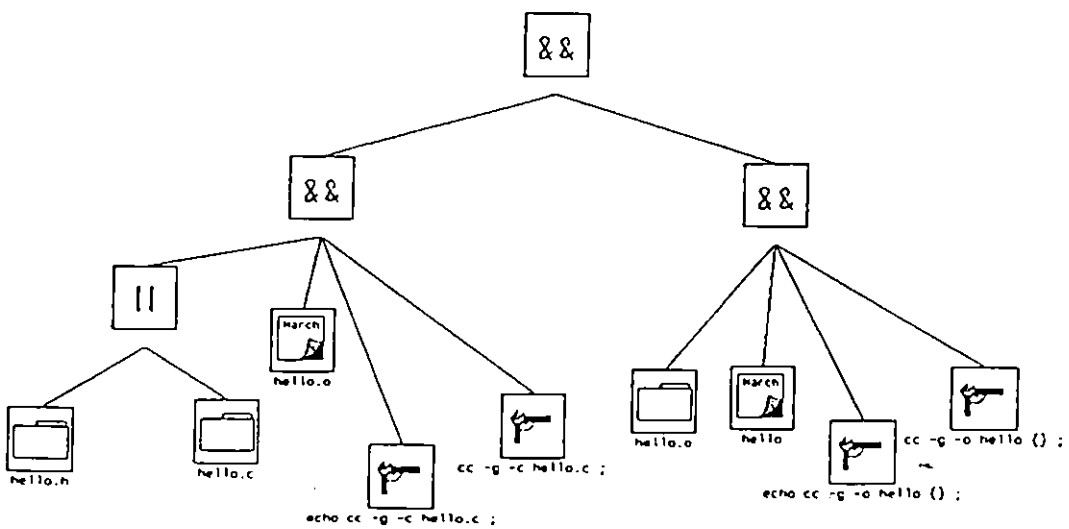


Figure 3.3: *make hello* (1)

This solution performs both the re-compilation and re-linking to rebuild `hello` and aborts if there are compilation errors. Granted, it cannot test for missing targets, but it has a far more glaring flaw — it is dependent on *find* presenting the source files from a directory before any object files. If `hello.c` precedes `hello.o`, it is re-

compiled producing a new object file to test against `hello` prior to re-linking. Unfortunately, Findtool has no control over the order files are retrieved from a directory and, therefore, work may be left incomplete. If `hello.o` is tested before `hello.c`, it is not relinked after a new version is re-compiled.

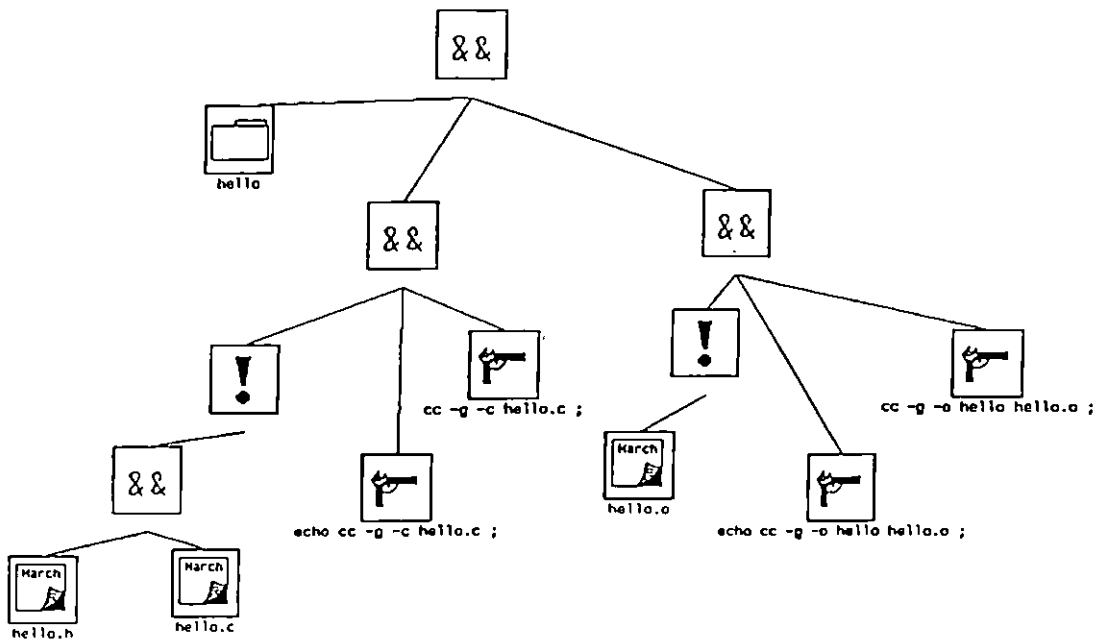


Figure 3.4: *make hello* (2)

Now let us take an opposing tack: the Findtool script in Figure 3.4 waits until the target `hello` is found. Then the time-stamps of the dependencies are checked. But since the roles are now reversed, the test results must be negated. The process proceeds directly through re-compilation and re-linking, unless interrupted by a compilation error. There is an implicit assumption that only source files are edited.

Since either `hello.h` or `hello.c` are newer than `hello`, `hello.c` is re-compiled to yield `hello.o`, and since `hello.o` is now newer than `hello`, a new `hello` is linked. Therefore, this script does not depend upon the order of the directory search. This script does require a small amount of assistance to complete its task: a target `hello` must exist before any re-compilation can occur (i.e., *touch* `hello`).

Let us embark on a more ambitious mini-project: *Makefile* 2. Its solution is depicted in Figure 3.7. Its clearest explanation is to demonstrate how it was designed in a top-down fashion.

Makefile 2

```
CFLAGS = -g -f68881
INC = surface.h polygon.h depth.h
OBJS = surface.o polygon.o depth.o
LIBRARIES = -lsuntool -lsunwindow -lpixrect -lm

surfer: $(OBJS)
      cc $(CFLAGS) $(OBJS) -o surfer $(LIBRARIES)

depth.o: depth.c depth.h polygon.h
      cc $(CFLAGS) -c depth.c

polygon.o: polygon.c depth.h polygon.h
      cc $(CFLAGS) -c polygon.c

surface.o: surface.c $(INC)
      cc $(CFLAGS) -c surface.c
```

The overall strategy is to apply the techniques learned while experimenting with *make hello*. Following that plan, a test for the target `surfer` is required, followed

by predicates to recompile each source module, then a linking predicate. Let us neglect any dependencies for this initial version shown in Figure 3.5.

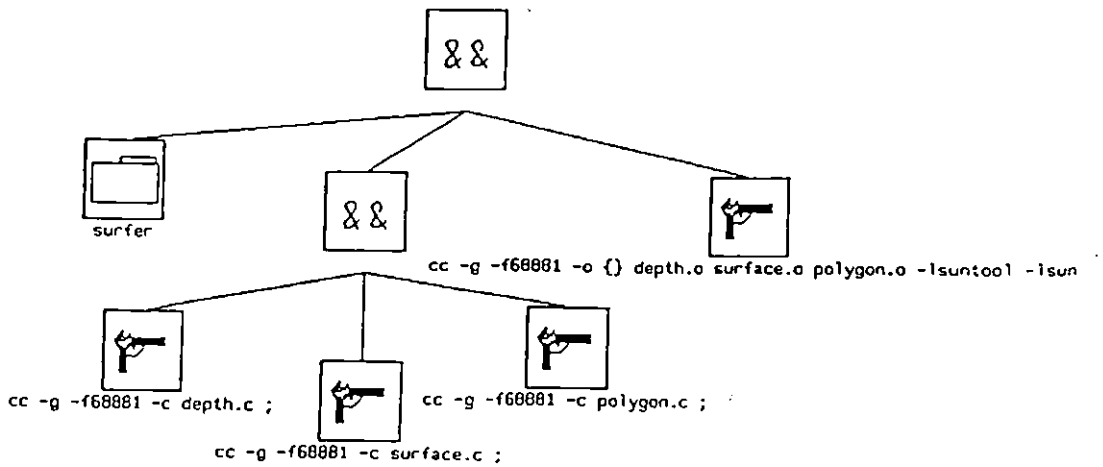


Figure 3.5: ***make surfer*** **(initial version)**

The technique used in the second *make hello* example is applied to this problem to express the dependencies for re-compiling the source modules. The Findtool sub-expression to re-compile `surfer.c` is shown in Figure 3.6. Similar expressions are developed to re-compile the other source files.

Why are the selected items in Figure 3.6 necessary? What would be the result of the Findtool expression if the selected items were removed? There are three possible outcomes:

1. `surface.o` is up to date, and thus no re-compilation is required, yielding false;

2. the re-compilation of `surface.c` fails, yielding false; and

3. the re-compilation of `surface.c` is successful, yielding true.

In cases 1 and 3 re-linking may be required, but the `-a` operator, which controls re-compilation, cannot be prevented from returning the false in the first case upwards in the abstract syntax tree. The boolean return code is insufficient for representing this condition so the Findtool expression is modified to always return false. The predicate chosen is "`-exec touch .compiled ;`", negated to produce a false result.

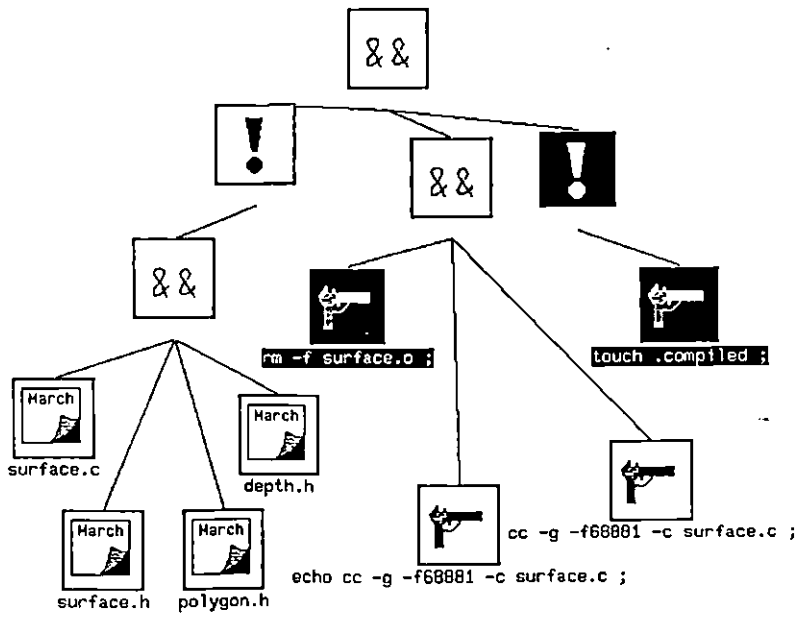


Figure 3.6: Findtool expression to re-compile `surface.c`

Now let us construct the complete Findtool expression depicted in Figure 3.7. The echoing `-exec` predicates have been removed for brevity.

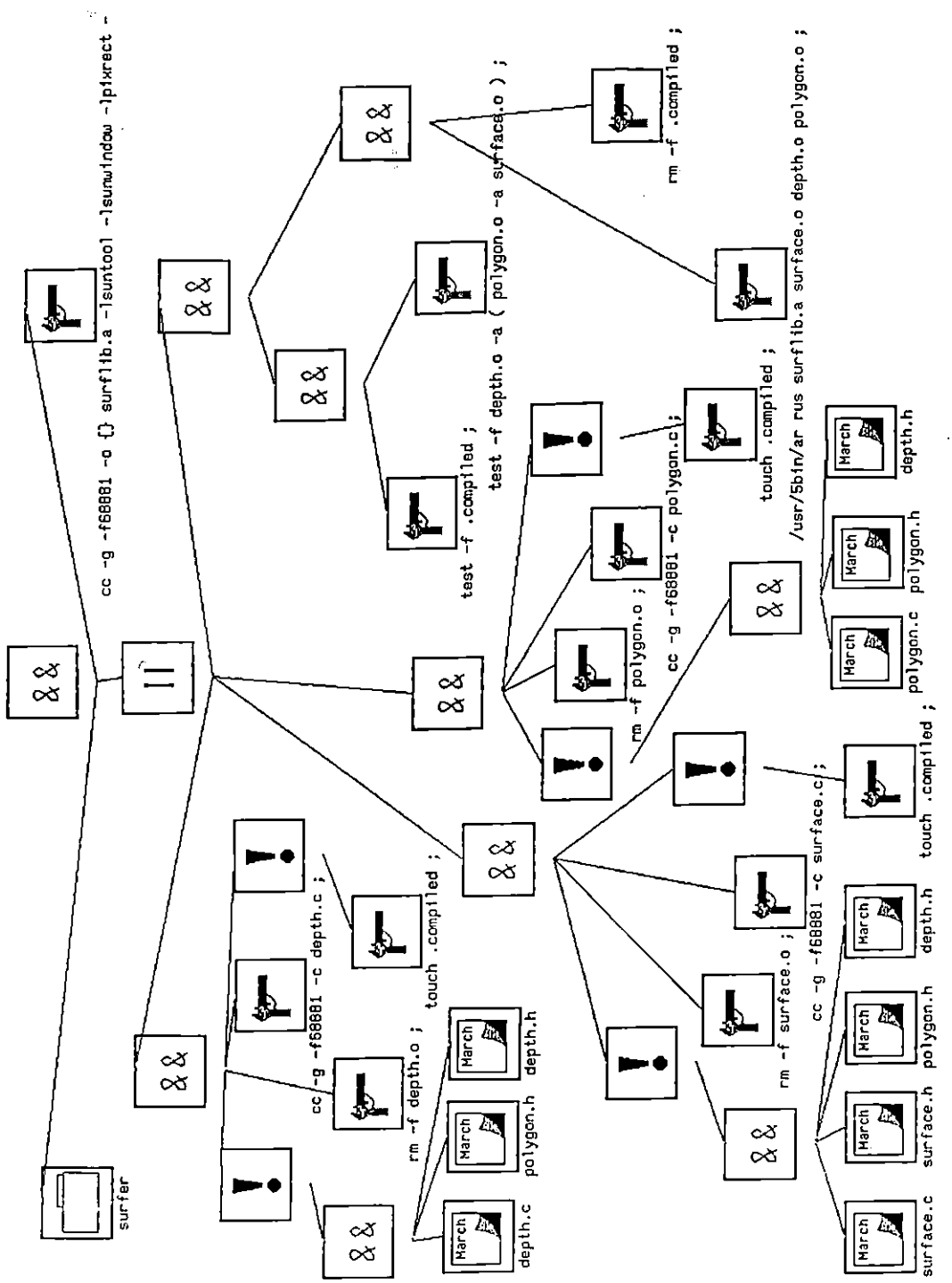


Figure 3.7: *make surfer*

Since all the re-compilation sub-expressions return false, it is correct to `-o` them together to check their dependency lists. However, because the result of "or"-ing a list of false boolean values is still false, a trailing expression is necessary to return true if re-linking is required. We take this opportunity to construct an archive library.

Re-linking should be performed when all object files are up to date. Therefore, an object file is removed before attempting re-compilation. If the object file is allowed to remain, it is impossible to determine, when building the archive library, whether re-compilation has failed. A *makefile* would not re-link unless some object files were updated on this pass. Hence, after successful re-compilation, the file `.compiled` is touched so that its presence may be tested later.

Fortunately, there appears to be a regular pattern for expressing compilation dependencies: additional dependencies would be added as sub-expressions of the central `-o` operator. If some modules require re-compilation ahead of others, they must appear to the left to be processed first.

There are some incidental features of *makefiles* that this Findtool script does not support:

1. The target `surfer` must be present for any re-compilation to occur;

2. Commands passed the system by `-exec` predicates are not echoed unless the programmer pairs an *echo* statement with each one explicitly;

3. *Makefiles* do not descend through the directory structure; hence all sub-directories should be pruned;

4. Even simple text substitution macros are not supported in Findtool; and,

5. *Makefile* 2 aborts at the first compilation error, as opposed to the Findtool script in Figure 5.7 which evaluates all dependency lists, attempting to re-compile all targets.

The Findtool expression is growing faster than its companion *makefile* because *find* interacts with the file system at a more primitive level. It is hardly a fair comparison of textual and visual programming environments. Perhaps *makefiles* could be improved with a graphical interface, but that would require an in-depth study, well outside the scope of this thesis.

The Findtool script depicted in Figure 3.7 translates to the *find* command shown in Figure 3.8. (For this *find* command script to perform properly, the `-newer` predicate bug must be fixed.) The implementations are equivalent. Which script is more aesthetically pleasing? Some programmers familiar with *find*'s syntax may opt for the standard *find* but the majority of UNIX users, inexperienced with *find*, should prefer constructing and maintaining the Findtool script.

3.5.3 Experience gained from simulating Maketool

The Maketool example in Figure 3.3 illustrates how programmers can be deceived into accepting inconsistent Findtool scripts, because there is no control over the order files are retrieved from directories. Programmers can specify a sorting function if they require uniform behavior from Survey scripts.

```
find . \
-name surfer \
-a \( \
     \( \
          \( \
               \( \
                    \! \(-newer depth.c -newer polygon.h \
                                   -newer depth.h \) \
                    -exec rm -f depth.o \; \
                    -exec cc -g -f68881 -c depth.c \; \
                    \! \( -exec touch .compiled \; \) \
               \) -o \( \
                    \( \
                         \! \(-newer surface.c -newer surface.h \
                                   -newer polygon.h -newer depth.h \) \
                         -exec rm -f surface.o \; \
                         -exec cc -g -f68881 -c surface.c \; \
                         \! \( -exec touch .compiled \; \) \
                    \) -o \( \
                         \! \(-newer polygon.c -newer polygon.h \
                                   -newer depth.h \) \
                         -exec rm -f polygon.o \; \
                         -exec cc -g -f68881 -c polygon.c \; \
                         \! \( -exec touch .compiled \; \) \
                    \) \
               \) \
          \) -o \( \
               \( \
                    -exec test -f .compiled \;
     -exec test '-f depth.o -a ( polygon.o -a surface.o )' \; \
               \) \
               -exec /usr/5bin/ar rus surflib.a surface.o \
                                   depth.o polygon.o \; \
               -exec test rm -f .compiled \;
          \) \
     \) \
     -a \
     \( -exec cc -g -f68881 -o '{}' surflib.a -lcore -lsuntool \
          -lsunwindow -lpixrect -lm \; \) \
)\
```

Figure 3.8: *make surfer find* command

The pattern-matching scripting language *awk* [AKW 79] has two pre-defined patterns `BEGIN` and `END` activated before any of the lines of the file are scanned and after the end-of-file has been read, respectively. Suppose Findtool creates two temporary hidden files (`.begin` and `.end`) when a directory is opened for the benefit of two new predicates, `-begin` and `-end`. Of course, these files would be removed when the directory is closed.

When `-begin` is called, a hidden file, say `.target`, could be created before any other processing occurs. As part of the target processing, `.target` is removed. Then, during the `-end` phase, if `.target` still exists all the sources are compiled and linked for the first time. There are many similar situations where these predicates are valuable within Survey scripts for initialization or cleanup.

Within Findtool, building relationships between icons is the only avenue explored in composing scripts. SIL-ICON is a visual language compiler for creating composite icons, incorporating the semantic actions of primitive icons [CTYY 89]. Composite icons would reduce the number of predicates in a Findtool expression, improving the allocation of display real estate.

One icon that appears frequently in the *makefile* scripts is the negation operator. For Survey's boolean operators, when the "*Negate*" menu item is selected, the operator is replaced by another which performs the identical test, but negates the boolean result. The negated operator's icon is annotated with an '!', signifying negation. Negating a negated operator restores the original.

Any Survey predicate with the potential to modify the Rigi database has a

"*Confirm*" icon menu item requiring a simple yes or no confirmation of the action. The icon requiring confirmation includes an '?'.

3.6 Summary

Findtool is an effective substitute for *find*. All the predicates of the original *find* are supported. Only the `-o` operator has been extended to accept an arbitrary number of sub-predicates and a bug in the `-newer` predicate was corrected.

Findtool is bound by many of the limitations of *find*. It cannot completely replace *make*, for instance, but it was never designed to do so.

The Findtool interface introduces the Rigi Editor interface as an effective visual programming environment for a simple scripting language like *find*. Since Survey is designed to mimic *find* within the Rigi Model, the Rigi Editor is expected to support its visual programming environment well too. Working with Findtool scripts permitted deficiencies in *find* to be detected and improvements to Survey to be suggested before it was developed.

4. A visual scripting language for Rigi

This chapter presents *Survey*, a visual scripting language for the *Rigi Editor* [Müll 86, MüKl 88]. It is assumed that the reader is familiar with *Findtool* and the SunView implementation of the Rigi Editor [Klas 88].

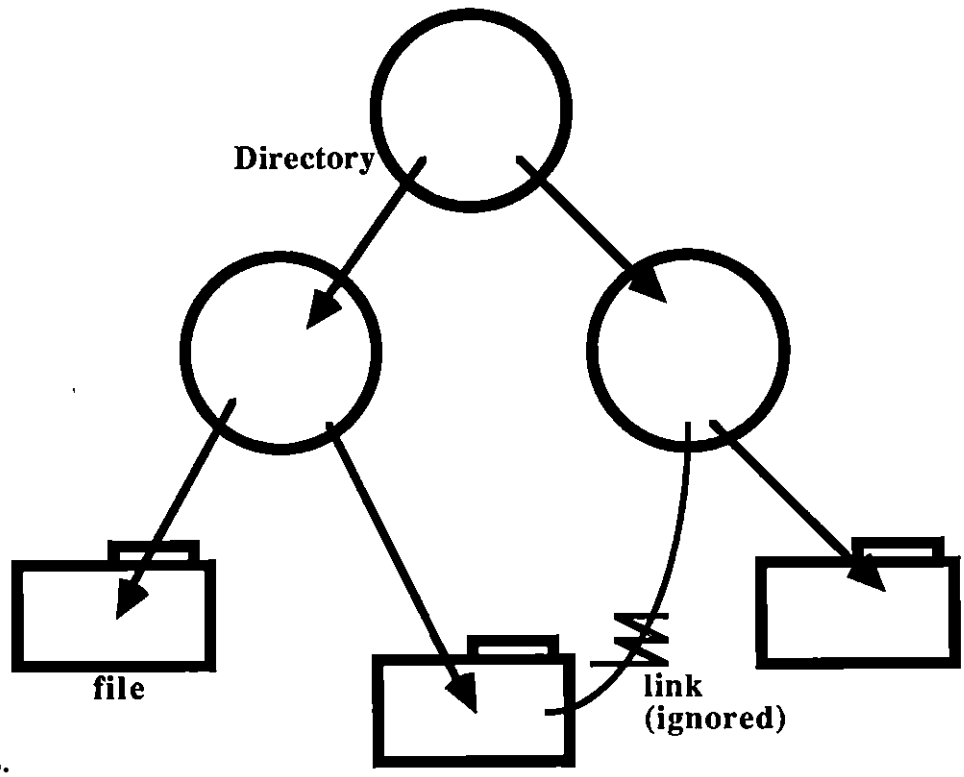
4.1 Introduction

To become truly useful as a software development environment, the Rigi Editor needs a *scripting language.* While basic project management tasks can be performed manually with the Rigi Editor, it is too tedious. Survey scripts assist in automating recurring tasks and reduce the possibility of careless errors.

Like its inspiration the UNIX *find* command, Findtool exploits the hierarchy of the UNIX file system to perform file maintenance. Is there not a remarkable similarity between the conceptual view of the UNIX file system is depicted in Figure 4.1 and the pyramid-like structure depicted in Figure 2.1, a conceptual view of the *Rigi Model* [Müll 86]? Intuitively, similar scripting power within Survey should be useful in the Rigi Editor for processing hierarchies in the Rigi Model.

The Rigi Model likens its presentation of a software development environment to the commanding view of the scenery from a mountain top. In expanding on this theme, 'View' or 'Viewpoint' would be good choices to name a scripting language for the Rigi Editor, but the names are too passive. 'Survey' incorporates the potential to modify the

Rigi database with command scripts.



scripts.

Figure 4.1: The UNIX Hierarchical File System

4.2 Lessons from Findtool

From the outset Findtool was designed as a prototype. Besides making *find* more convenient to use, Findtool demonstrated that the Rigi interface standards could support editing of command scripts.

Some design changes were necessary to convert Findtool into Survey:

1. New, Rigi-specific predicates, more appropriate to Survey, have been defined and Findtool's UNIX-specific predicates have been dropped;

2. The experience gained programming Findtool scripts demonstrates the need for the `-sort`, `-begin`, and `-end` predicates to control the behavior of scripts;

3. Survey predicates can be negated without introducing a second predicate;

4. Findtool was designed to process a hierarchy of files stored in the UNIX file system, whereas Survey processes a hierarchy of objects stored in the Rigi database;

5. Like UNIX files, Rigi objects are typed and identified by name, but a name is not necessarily unique within the scope of a Rigi graph;

6. Findtool ignores symbolic links, leaving directories as the only type of link, but Survey differentiates between the several types of arcs within the Rigi Model; and,

7. Findtool allocates a special text window for displaying its output, while Survey directs its output to `stdout` and `stderr`.

4.3 Running Survey

The Rigi Editor handles all preparations for using Survey. Survey is invoked by opening a Survey icon. The Survey's iconic interface behaves exactly like the Findtool Editor. An active Survey session is depicted in the foreground window of Figure 2.2.

Unlike Findtool, scripts are not executed from within the Survey. Instead, using the selection mechanisms of the Rigi Editor, a set of Rigi icons is selected. If the selected set includes a single Survey icon, then when the Survey icon is selected from the accessories document menu, the Survey script is applied to all other Rigi objects in the selected set. If there is no selected set when the Survey icon is selected, a fresh

Survey script is created.

Incorrect syntax causes Survey to generate Rigi alert boxes. As in Findtool, the error message is displayed until acknowledged. The Survey error messages are listed in Appendix C.

4.4 Survey predicates

4.4.1 Justification

A scripting language like Survey adds tremendous flexibility to the Rigi Editor by making it *extensible*. An extensible application provides the capability to build new functions upon its basic primitives.

Given the objective of designing an extensible application, it is essential to identify the primitive operations of the Rigi Editor. It is vital that the extensibility of Survey not be constrained needlessly by omitted predicates. The Survey predicates are grouped into five categories:

1. Node status — predicates to query the state of a given node;

2. Arc status — predicates to query the status of arcs joining nodes;

3. Arc manipulation — predicates to change the relationships among nodes;

4. Clipboarding — predicates to manipulate the selected set and the clipboard; and

5. Control — predicates to structure the abstract syntax tree and control the processing of Rigi objects.

Since *find* is imbedded within the UNIX command *shell*, a `-exec` predicate is permissible for Findtool, but because Survey users have no idea of how Rigi objects

are represented within the Rigi Editor, Survey cannot offer this kind of extensibility. Instead, each function applied to the contents of a Rigi object must be supplied through a Survey predicate. As a compromise, the functions most essential to Rigi's objective as a software development environment are provided first: opening (viewing), removing, duplicating, printing, compiling (or building, if appropriate), and searching of objects. This area is likely to be influenced by future upgrades: a stream editor might only be one example of a function added at a later date.

The remainder of this section briefly summarizes the Survey predicates. The following conventions for expressing parameters are adopted from Findtool:

1. The standard *shell* metacharacters (i.e., '*', '?', and '[]') apply to Rigi object names.

2. The argument n represents a decimal integer where '+n' means "greater than n," '-n' "less than n," and 'n' "exactly n."

Parameters are entered as in Findtool (cf. Appendix B). Survey predicates accept either no parameters or static parameters, entered in the icon's name field. No Survey predicates use a SunView text editor pop-up.

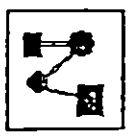
In addition, certain Survey predicates maintain an internal state. The initial state of the `-contents` predicate is "do not designate any node contents." The `-showarc` and `-drawarc` predicates take their initial values from the current settings of the Rigi Editor.

The node types supported by the Rigi Model are: *sys*, *mod*, *def*, *imp*, *gen*,

alt, *pro*, *rev*, *dat*, *doc*, and *pic*. A node may support some or all of these documents as contents: *source*, *info*, and *self*, where 'self' refers to its sub-graph.

The dependencies represented within the Rigi Model are: *structure*, *change*, and *semantic*. The Rigi Editor displays arcs according to the following scales: 'None', 1, 2, 4, 8, 16, 32, and 64. In addition, the `-showarc` predicate supports the 'Normal' option displaying arcs unfiltered (i.e., according to the dimensions they were inserted).

4.4.2 Node status predicates



-graph object

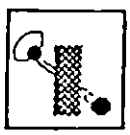
true if the node is contained in the graph "object"

-name object

true if the node's name is "object"

-echo

print the name of the node (always true)



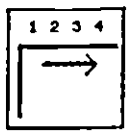
-visible x

x > 0 make the node visible (true)

x = 0 true if the node is visible

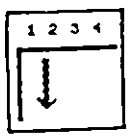
x < 0 make the node invisible (true)

The `-visible` predicate applies to all nodes of the same type as the current node.



true if the node's x-coordinate is n

-x n



true if the node's y-coordinate is n

-y n

true if the node is of the specified type

-type type

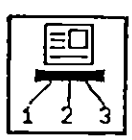
4.4.3 Arc status predicates

show arcs of type "arctype" in size "size"

(always true)

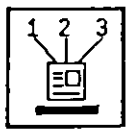
-showarc arctype size

An arc must be 'showing' (i.e., its size must be positive) for the arc to be detected by these arc status predicates.



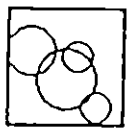
true if the node has n successors

-succs n



true if the node has n predecessors

-preds n



true if the node has n nodes linked to it

-links n

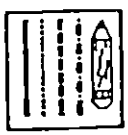
true if the node has a predecessor named "object"

-parent object

true if the node has a successor named "object"

-child object

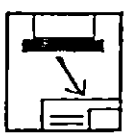
4.4.4 Arc manipulating predicates



arcs inserted will be of type "type" and size "size"

(always true)

-drawarc arctype size



insert an arc from the node to successor "object"

(true, if successful)

-succ object

insert an arc from predecessor "object" to the node

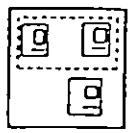
(true, if successful)

-pred object

In keeping with the interface conventions established for Rigi, if `-succ` or `-pred` is requested to re-draw an already existing arc, the arc is deleted.

4.4.5 Clipboarding predicates

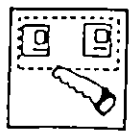
x > 0 add node to the current selection (true)



x = 0 clear the current selection (true)

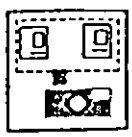
x < 0 remove node from the current selection (true)

-select x



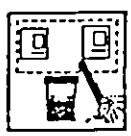
'cut' the current selection (always true)

-cut



'copy' the current selection (always true)

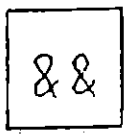
-copy



'paste' content of the clipboard (always true)

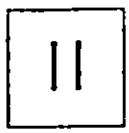
-paste

4.4.6 Survey control operators and predicates



true if all sub-predicates are true

-and

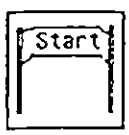


true if any sub-predicate is true

-or

true if sub-predicate is false

-not

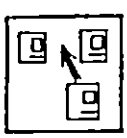


true if the graph has just been opened (before any nodes from the graph have been processed)

-begin

true if the graph has just been closed (after all nodes from the graph have been processed)

-end



sort objects in graph into alphabetic order by name before processing (always true)

-sort

abort processing of graph (always false)

-prune

process Rigi database depth first (always true)

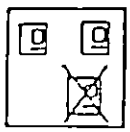
-depth

4.4.7 Node processing predicates

open[+] or close[-] the 'contents'

true if performed, false if no document is available

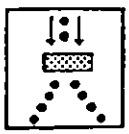
-open [+ | -]doc



remove (delete) the node from the current graph

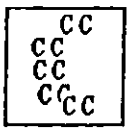
(true if successful)

-remove



duplicate the node (always true)

-dup



compile the source document

true if no errors are detected

-compile

link the application

true if no errors are detected

-build

+ include (- do not include) the document in processing the object's contents (always true)

-contents [+ | -]doc

The following predicates apply to the `-contents` specified above. It is the programmer's responsibility to ensure that the predicates are applied to suitable contents within a node.

list the node 'contents' on the "printer" (always true)

-print printer

Search the 'contents' of the node for any matching string (true if any lines are found)

-grep -options "string"

Options are passed to *grep*. If no options are supplied, '-' is still required.

The string data must be in quotes.

4.5 Survey examples

4.5.1 Used-by relation 'explosion'

Suppose that a programmer, in coding a module `survey.c`, names a function `select`, creating a name conflict with the routine in the system library. C supercedes

a library routine at a programmer's discretion. Later, the programmer discovers the error and decides to rename the function as `select_set`. Not only must the source file, `survey.c`, be updated, but its header, `survey.h`, and all occurrences of `select` in other modules need to be examined.

For every module its *uses relation* includes the set of all the modules supplying resources to it. Here, the programmer needs to compute the module's *used-by relation* (i.e., the set of all modules using `survey.c`). A Survey script is a powerful alternative to computing the used-by relation manually. Let us suppose that the programmer would like to open each source document referencing `select` for editing. Depending upon how frequently `select` is used, there could be an 'explosion' of windows being opened.

A `-grep` predicate is included in Survey to search the contents of Rigi objects for key strings. Like the UNIX *grep* utility, the Survey `-grep` predicate does not recursively descend through the Rigi database. A Survey script, similar to the rgrep example introduced in Section 3.4, can be programmed to recursively search Rigi hierarchies.

Within Rigi, the uses relation for an object is defined by the *structural dependency* arcs emanating from it. Then the used-by relation is given by the object's incoming structural dependency arcs. Therefore, the programmer requires a Survey script which opens objects that have `survey.c` as a structural successor, containing a call to the function `select`.

Figure 4.2 depicts such a Survey script. The script consists of three sub-

expressions: the initialization of Survey's internal status variables to search the source documents and show (and query) structural dependencies; a test of the incoming structural dependency arcs; and, if those modules contain the keyword "select," open the source document for editing.

Source code is maintained exclusively in *rev* objects. Yet the structural dependency arcs of interest are between *sys, pro,* and *mod* objects higher in the Rigi hierarchy. To speed up the search, the programmer could select those objects that are structurally dependent upon `survey.c` for Survey to process. But in general, the script needs to `-prune` those objects which are not a (structural) `-parent` of `survey.c`. (That is also expressed as objects without a `-child` called `survey.c`).

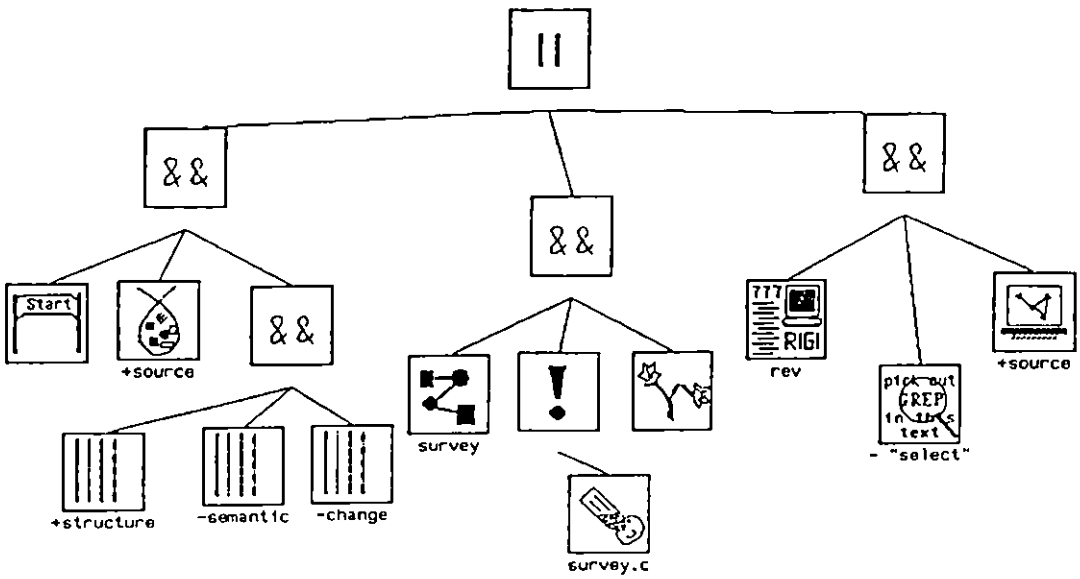


Figure 4.2: Used-by relation 'explosion' Survey script

Unfortunately, all revisions of a module using `select` are opened. The Rigi Model could be redefined to link the most recent revision directly to the module object.

Referring to Rigi objects by name can produce unexpected results because the names of Rigi objects are not necessarily unique, even within graphs. Naming conflicts must be avoided when using Survey scripts.

4.6 Summary

Survey offers the advantages of "user-friendliness" and visual support for creativity in a programming environment. As part of the Rigi Editor, it is crucial that Survey espouse the same editing standards: Survey scripts look like Rigi dependency graphs and are manipulated in the same manner.

It was never intended that Survey should completely replace the C programming language for all tasks. For instance, the 1500 lines of C code that implement the cleanup subsystem is far too ambitious to be attempted within Survey. However, Survey scripts can serve as simple prototypes of functions before they are translated to C for performance considerations.

A scripting language is essential for the Rigi Editor to be an effective software development environment. Survey command scripts permit tedious, error-prone procedures to be pre-programmed more reliably and executed with less effort. Nevertheless, some inconvenient limitations are encountered which could be removed with further design work.

Survey scripts do not support parameters or local variables. The Rigi Editor's selected set and the clipboard act as global variables, but it is desirable to define private selections and test for membership. Parameters would allow Survey scripts to repeat the processing of similar Rigi objects without re-editing. Survey scripts cannot be called as functions, but without parameters there is not much point.

Find is well-suited to processing hierarchies of the UNIX file system; other *shells* are designed for tasks where recursive descent is not beneficial. When an object has been located within the Rigi Editor, four relationships can be investigated: a successor, a predecessor, the sub-graph, or the super-graph. Descending to the sub-graph is arbitrary when the searching strategy could be specified with four predicates.

5. A reusable system design

Survey is an amalgam of the functionality of the UNIX *find* command, the *Rigi Model* as a software development environment, and the interface standards of the *Rigi Editor*. As an intermediate step, *Findtool* integrates the Rigi style interface with *find*. Programming and system design issues are the focus of this discussion; in particular, a reusable system design is identified.

5.1 Introduction

5.1.1 A design prototype is necessary

Survey is a scripting language, modelled after *find*, to process hierarchies in the Rigi database. It employs the Rigi Editor interface as a visual programming environment. To support these capabilities Survey requires:

(a) the Rigi Editor interface, customized as its programming environment;

(b) the graph structures of the Rigi Editor;

(c) the Rigi database and the ability to traverse its hierarchies;

(d) a scanner and a parser for its scripts;

(e) a "globber," to compare strings containing meta-characters; and,

(f) code to implement its predicates.

Except for the last element in the list, no functions present in Survey are not found in either *find* or the Rigi Editor in some form. One would expect to reuse

software from these other projects in this development.

But, what software? Is it worth reusing? The three key issues in reusing software are: *finding* software to reuse; *understanding* it sufficiently to reuse it effectively; and, being able to *modify* it, if necessary. Software without these qualities cannot be reused.

Findtool was proposed as a prototype of Survey to investigate the suitability of the Rigi Editor as a visual programming environment. The prototype attempts to reuse software from *find* and the Rigi Editor to assess its suitability for reuse in Survey. Reusing software in a prototype develops understanding and experience modifying it.

5.1.2 Identifying a generic system design

Suppose that developing Findtool isolates some reusable software. There is no guarantee that this software will be useful in Survey, unless the development of both Findtool and Survey are operating under the same overall strategy. A generic system design, reused in both projects, provides this structure. All software systems are constructed from basic services. A *generic system design* identifies the common services required by a class of related systems. Thus, we can analyze the services required in both Findtool and Survey to identify a generic system design.

The services required in Survey are listed in Section 5.1. The services required to support Findtool are:

(a) the Rigi Editor interface, customized as its programming environment;

(b) the graph structures of the Rigi Editor, to store Findtool expressions;

(c) the UNIX file system and the ability to traverse its hierarchies;

(d) a window to intercept *find* command output;

(e) a scanner of Findtool expressions;

(f) a parser of *find* expressions;

(g) a "globber"; and,

(h) code to implement the Findtool predicates.

There are many services common to Findtool and Survey. A generic visual programming environment, exploiting the Rigi Editor interface, for a scripting language to process hierarchical data structures requires:

(a) a customized version of the Rigi Editor interface;

(b) the graph structures of the Rigi Editor;

(c) the ability to traverse hierarchical data structures;

(d) a scanner and a parser;

(e) a "globber"; and,

(f) code to implement its predicates.

In developing Findtool, the objective was to locate and understand the software providing these essential services.

5.1.3 Design for change fosters reuse

In order to reuse this generic system design in Survey, some software will be modified to accommodate the requirements of Survey that differ from Findtool's.

These changes in the delivery of some services should be anticipated and reflected in the module decomposition. It is important to manage changes by restricting them to the internals of modules [Parn 72]. In Findtool, a module supplies a service, encapsulating data structures and algorithms necessary to deliver its service, enabling them to be changed independently from the rest of the system. Even when the internals of a module change, the service provided remains the same, permitting reuse of its interface.

Variants of a module share the same interface but supply different implementations. Variants of the generic system's services are reused throughout Findtool and Survey. Variants are easier to understand and verify because changes are made to the implementation dependent portion of a module, leaving the specifications of its interface constant. The previous implementation can offer guidance.

5.2 Synthesizing Findtool

Findtool is an amalgam of *find* and the Rigi Editor. The Findtool Output Window is the only service required by Findtool, not provided by *find* or the Rigi Editor.

Through developing Findtool, the services essential to the generic system are finalized. Once the software presented for reuse is understood, independent modules to support the generic system design are identified or created, if necessary. Then, because Findtool is itself a variant of the generic system, it is implemented reusing generic services, or by creating variants of them.

5.2.1 Find command module structure

Find was implemented by Woods [Wood 81]. While the code is contained within a single source file, an implicit module structure is readily apparent throughout. Since a module guide, describing these modules and their services, is not available, a sketch of one is outlined below.

1. Scanner

Scan the command line sequentially to isolate tokens in the input stream.

2. Parser

Parse the scanned tokens to form an abstract syntax tree. Detect syntax errors.

3. Predicates

Implement the predicates for *find*.

4. Globber

Perform "globbing."

5. Traversal

For each pathname in the list, and for each file or directory under that path in the UNIX file system hierarchy, perform a recursive descent through the file structure, applying the abstract syntax tree built earlier for each file or directory found.

6. Command Editor

Woods does not write any code to compose *find* commands because he assumes that any UNIX system provides a *shell* or editor for this purpose.

5.2.2 Rigi Editor module structure

The implementation of the Rigi Editor is the topic of Klashinsky's M.Sc. thesis [Klas 88]. It serves as an excellent module guide for the software. Nevertheless, a summary of the key subsystems is discussed here. Some subsystems are ignored because they are not relevant for the integration of Survey into the Rigi Editor.

1. Graph Subsystem

The *graph* subsystem creates and manipulates graphs maintained in memory within the Rigi Editor. It is sub-divided into three modules: the *node* module, for managing nodes; the *arc* module, for maintaining arcs; and the *graph* module, for collecting the nodes and arcs into graphs.

2. Menu Subsystem

SunView encourages the use of pop-up menus to interact with the user. Rigi follows this convention with walking menus triggered by a right-press on each node as well as the *canvas* background. The *menu* subsystem is split into the *node menu* module and the *background menu* module.

3. Popup Module

Pop-ups are alerts to bring information to the user's attention (i.e., warn of error conditions, or prompt for confirmation). The message is displayed until acknowledged with a mouse click.

4. Database Subsystem

The *database* module accesses graphs stored in *ndbm* (a UNIX database

package [SUN 88a]) databases. The *dbnode* and *dbgraph* modules map the data structures of the *graph* subsystem into records suitable for the *database* module and vice-versa.

5. Cleanup Subsystem

The *cleanup* subsystem re-draws Rigi graphs in an aesthetically pleasing format.

6. Edit Module

The current selection and the clipboard are managed by the *edit* module. Arcs are inserted and deleted to support the cut, copy, paste, and clear operations.

7. Canvas Module

The *canvas* module is the event handler for graph windows containing drawing canvases.

8. Imaging Module

The *imaging* module defines the visual representation of Rigi objects.

9. Window Module

A UNIX process is allocated a finite number of file descriptors. Each SunView object consumes file descriptors. Because SunView does not handle the exhaustion of file descriptors elegantly, all requests for SunView objects are shunted through the *window* module.

5.2.3 Creating the generic system services

The generic system design requires the system services identified in Section

5.1.2. A set of these system services is prepared by reusing the software surveyed.

For instance, the Rigi Editor interface is supported by: the *canvas* module (`mycanvas.c`), the *edit* module (`edit.c`), the *imaging* module (`images.c`), the *menu* subsystem (`canvas_menu.c` and `file_menu.c`), the *popup* module (`popup.c`), the *graph* subsystem (`graph.c`, `nodes.c`, and `arcs.c`), the *database* subsystem (`dbase.c`, `dbnode.c`, and `graphdb.c`), and the *cleanup* subsystem. The *cleanup* subsystem must be eliminated from the generic Rigi Editor interface because left-to-right ordering is not preserved when tidying Rigi graphs. What may not be apparent until the Rigi Editor software is clearly understood is that any system using the Rigi Editor interface requires the *window* module (`window.c`).

Many generic services are handled by the *find* source code: the scanner, the parser, the "globber," the traversal of hierarchical data structures, and the implementation of predicates. Because it is essential to have the capability of varying generic services independently, the *find* source code was split into independent modules, each comprising a single compilation unit. Thus, `find.c` spawns: `find.c`, `findcpio.c`, `finddesc.c`, `findglob.c`, `findscan.c`, and `findtree.c`. Because the scanner and parser need to function independently they are re-designed giving the scanner sole access to the input.

5.2.4 Building Findtool

When all the services to support the generic system design exist, Findtool is implemented as a variant, reusing the generic system design, creating variants of

services that change. For instance, *find* has a scanner to pluck tokens from the command line; under Findtool, this module retrieves tokens from the graph of the abstract syntax tree. They are variants because Findtool's scanner functions through the same interface, inserting the parentheses necessary to parse the Findtool expression.

Figures 5.1 provides a graphic synopsis of the generic software services reused as is (i.e., requiring atmost re-compilation) and the variants created.

5.3 Experience with prototyping Survey

Findtool provided motivation to develop a generic system design for Survey. Just as experimenting with Findtool demonstrated weaknesses in its interface that were rectified in Survey, developing Findtool lead to refinements to the generic system design.

5.3.1 Handling SunView events with a finite state machine

Klashinsky split the hypertext functions of the Rigi Editor across two modules: `edit.c` manages cut-copy-paste editing and operations on the selected set whereas `mycanvas.c` deals with SunView events. Findtool and Survey require these basic interface primitives and therefore these modules should both be reusable as is. `edit.c` was never changed and was linked as object code into Findtool.

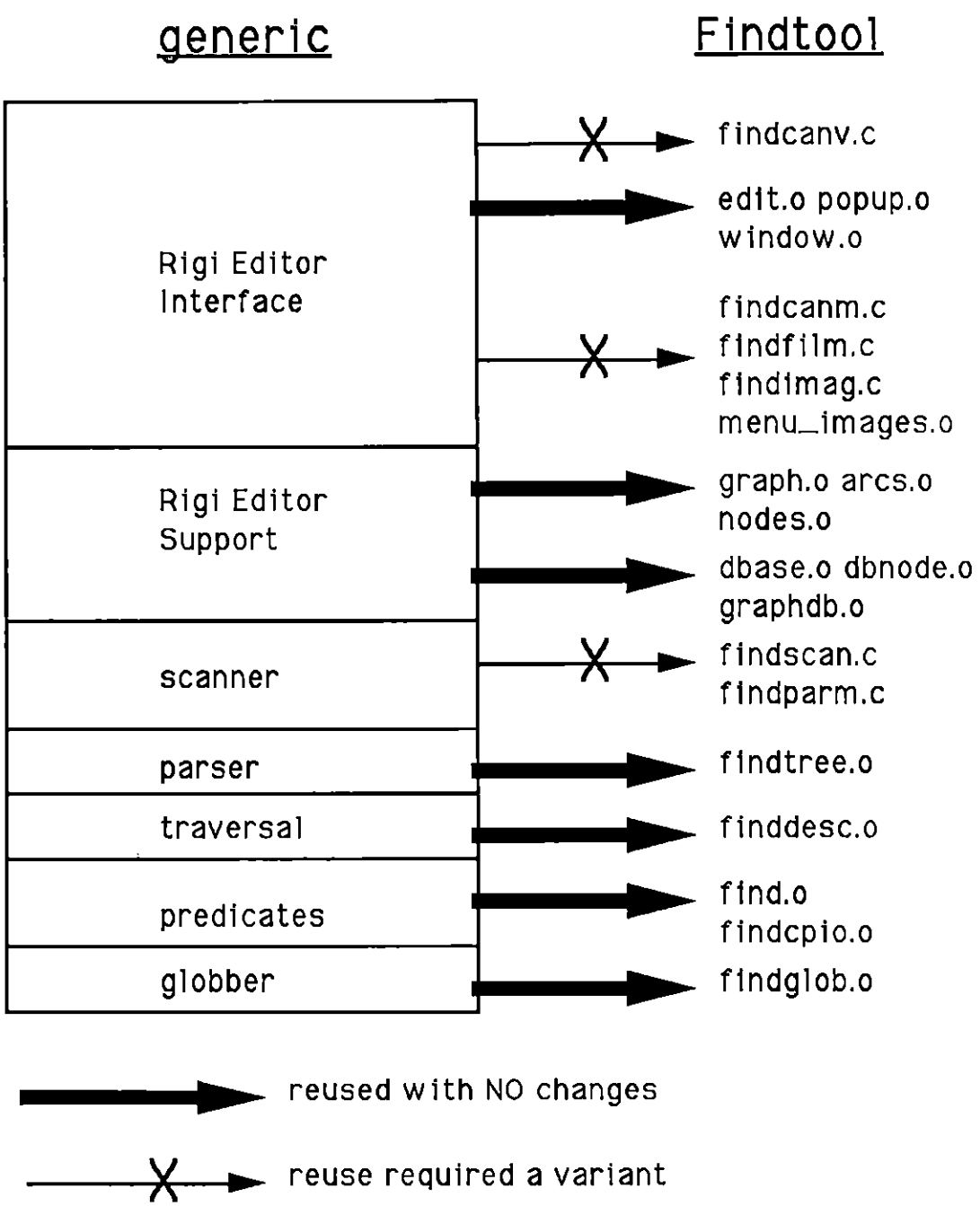


Figure 5.1: Reuse of generic services in Findtool

However, `mycanvas.c` was revised extensively in producing Findtool to remove code specific to the semantics of the Rigi Model. For instance, there is an extensive case statement activated when a menu selection is made. While this code has a functional relationship to event handling, it actually belongs to the menu definition module because each action corresponds to a given menu item.

Where did this miscellaneous code come from? Many experiments were conducted with the Rigi Editor interface during its early development. The objective was to try new hypertext editing paradigms, rather than creating a reusable software component. When others are waiting, there is tremendous pressure to make fixes as soon as possible. The cumulative effect of all these small changes introduced the semantics of the Rigi Model into the event handling, diminishing its reusability.

The interaction between a user and a computer program is no different than any other kind of communication involving *finite state automata*: a system is in one of a given number of states when it receives a stimulus from its environment. It responds in a prescribed manner, transferring to a new state.

Handling SunView events is intricate: the response to a given event may be dependent on the previous event. Klashinsky invented a custom event handler that attempts to handle all events in every circumstance. He chose a cumbersome manner to implement the finite state machine. Because of the convoluted structure of the event handling code, bugs are found as users venture through untried paths in the code. It is logically simpler to install another event handler representing the state transition.

Debugging and maintenance are simplified because each of the event handlers can be verified more easily than the omnibus event handler they replace. The new module structure involves some repetitious coding, but that is justified, because of the overall clarity in the structure of the implementation. Local functions collect common code.

5.3.2 Storing and retrieving predicate parameters

There is no difficulty storing or retrieving parameters for *find* because UNIX provides the command line. But, where can Findtool store parameters within Rigi objects? The name field is sufficient for most predicate parameters, but the `-exec` and `-ok` predicates may require more characters. The redundant Rigi source documents are available.

If only the scanner needed to know the location of these parameters, when retrieving them, the representation of these parameters could be encapsulated entirely within that module. But, the parameters are editable as well. Once the technique of installing state event handlers is mastered, it is a simple matter to install an event handler from the *parameter access* module, `findparm.c`, to receive the parameter when the appropriate menu item is selected. Because this module bridges the Rigi Editor interface and the scanner, it becomes part of the generic system design.

5.4 Synthesizing Survey

With Findtool completed, a transition point is reached. Findtool stands on its own merits as a visual programming environment for *find* scripts. However, our goal

is to produce Survey, a similar scripting language within the Rigi Editor.

The development of a generic system design enabled Survey to be implemented as another variant to that design. As with Findtool, certain generic services are reused without change and some required a variant. Figure 5.2 depicts the reuse of generic system services in Survey.

For Survey there is an additional question to be answered, however. Which of the two variants of the *menu* subsystem should be used as the working implementation to base a variant? Obviously, the version that can be modified at least cost while producing reliable code: `canvas_menu.c` and `file_menu.c`, in this case, because Survey exists within the Rigi Editor, where the original menus are required.

5.5 Was it worth prototyping Survey?

Prototyping Findtool produced a revision to the generic system design better tailored to supporting visual programming environments based on the Rigi Editor.

But costs were accrued in building the prototype and designing the generic system that enabled software to be reused easily. Extra analysis was required to prepare the generic system. Some variants created for either the generic system or Findtool were not reused in Survey:

(a) the canvas event handler, `mycanvas.c`, containing Rigi Model semantics was unacceptable for reuse;

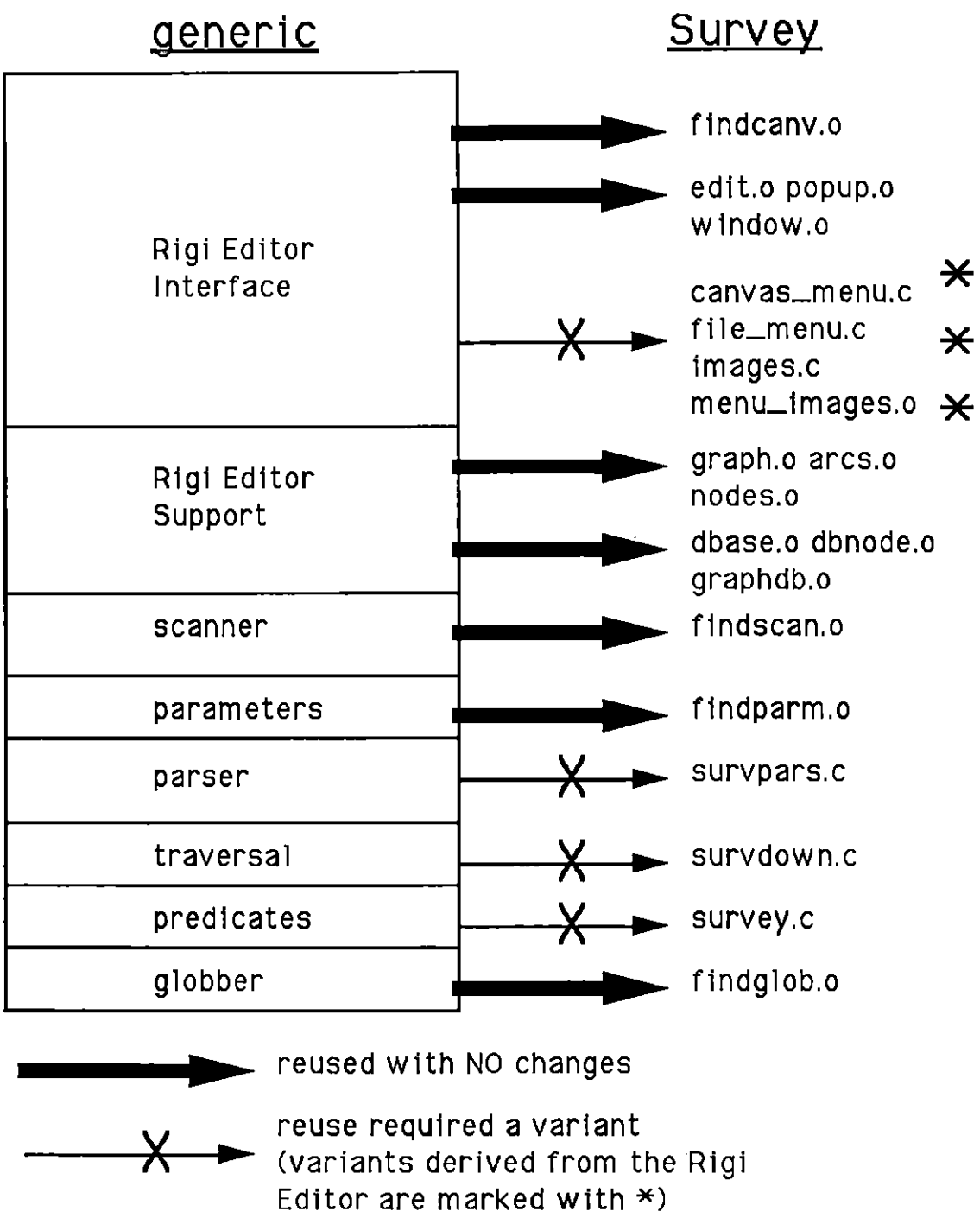


Figure 5.2: Reuse of generic services in Survey

(b) the Findtool parser, `findtree.c`, built an expression tree containing Findtool predicate semantics;

(c) the custom menus and icons created for Findtool, `findcanm.c`, `findfilm.c`, and `findimag.c`, were integrated back into their respective Rigi Editor variants for Survey;

(d) the traversal service was changed to process hierarchies in the Rigi database; and,

(e) the Survey predicates were not implemented until needed.

Despite these drawbacks, Findtool was still a useful design prototype. Large amounts of software from the Rigi Editor (roughly 2700 source lines in total) were reused in both Findtool and Survey after only re-compilation. While line counts make no allowance for program complexity, few programmers would neglect to pursue an opportunity to reuse this much debugged code pertinent to a project.

None of this software could be reused for free. Findtool offered an opportunity to understand how to use the software sufficiently before reusing it in Survey. Because the *menu* modules, the *imaging* module, the *parser* module, and the predicate implementation module all follow regular formats, they were easier to modify as they became more familiar.

It is not appropriate to assess the cost of these modifications to prototyping:

1. Any code reused from *find* would have had to be pared from its source file. Creating independent modules physically prepared the software services contained

within *find* for reuse and increased the understanding of the software.

2. Removing the Rigi Model semantics from `mycanvas.c` had been on the agenda of the Rigi project for some time. But the task never received high enough priority, since the hypertext editing features of the Rigi Editor had never been used independently from the Rigi Model.

While some effort was devoted to prototyping fruitlessly, it was more than offset by the volume of software identified for reuse, the experience gained with the software reused, and the skills developed in creating variants. The costs of prototyping Survey were controlled by minimizing the number of extraneous services in Findtool, anticipating changes to the generic services in creating variants, and constructing a large portion of the generic system from existing software.

5.6 Summary

In developing Survey, Findtool served as a prototype to investigate software available for reuse. To ensure that software developed through prototyping would be reusable in Survey, a generic system design was proposed. Findtool and Survey were implemented as variants of that generic system.

The success of a prototype in creating reusable software is dependent on the cost of identifying generic system services and creating their variants and the amount of reusable software produced in relation to the amount invested in prototyping. The successful use of variants is predicated on a design for change module decomposition philosophy. The role of the prototype in learning how to use and modify software

cannot be neglected.

A general event handler, independent of the Rigi Model semantics, was vital. Otherwise, separate event handlers would have been required for the Rigi Editor, Findtool, and Survey (within the Rigi Editor). The hypertext editing service was essential to all three tasks and should not be duplicated in all three designs. Severe maintenance problems would probably be encountered in keeping all three of these editors synchronized as the interface standards evolve.

6. Conclusions

6.1 Effectiveness of Findtool

The purpose of Findtool is twofold: first, to provide a superior interface to the *find* command on SUN Workstations; and second, to serve as a preliminary investigation of technical issues prior to the design and development of Survey as a scripting language for the Rigi Editor. Its effectiveness may be measured against either of these criteria.

Users of SUN Workstations will find that Findtool is an improvement for composing and executing *find* scripts. Findtool could act as a training environment until *find* is mastered. Because the "painful" *find* syntax is such a poor example of a command line interface, it is difficult to state emphatically whether Findtool is a testimonial to the effectiveness of visual programming environments in general.

Findtool improved the Survey scripting language by identifying deficiencies in its design that were corrected before implementation. Reuse of software and vital system design information was possible because constructing Findtool as a prototype produced a thorough understanding of the software involved and a generic system design that maximized the opportunity for reuse throughout the project.

6.2 Effectiveness of Rigi command scripts

With Survey scripts to program its interface, the Rigi Editor becomes an

extensible application. The flexibility of an extensible application will contribute to the longevity of Rigi as a software development environment because it can be programmed through Survey to meet new requirements more easily. Therefore, it is essential that all attributes of Rigi objects can be tested and modified within Survey. Survey is designed so that new predicates can be added to the language easily. Survey scripts can serve as simple prototypes of functions before they are translated to C for performance considerations.

Future design work could improve Survey as a scripting language. Survey scripts do not support parameters, local variables, or function calls. The recursive descent through the Rigi database is not always beneficial.

6.3 Critique of software reusability

6.3.1 Meeting the challenges of software reusability

Changeability and understandability are two of the most crucial qualities of a reusable software design. Within Rigi there has been a commitment to modelling software as objects that fulfill roles within the system. All data structures and algorithms (that are likely to change) related to a module's responsibility are encapsulated within the module. Consistent standards for coding and documentation are evident in the previous programmers' work on the Rigi Editor. A module will probably be accepted for reuse where it fits closely enough, even if some modifications are required.

What is unknown upon the completion of this investigation is the effectiveness of the presentation of design information within the Rigi Editor. Because important design decisions were reviewed by a committee of Dr. Müller's graduate students, some of the Rigi Editor software was already understood before design commenced. Nevertheless, the Rigi Editor software was much better designed for reuse than the *find* source code — it had to be hand separated into six sub-modules so that functionally independent parts of it could be reused. It was worth the effort to glean a thorough understanding of the modules in the absence of any documentation.

6.3.2 Reusing system designs

Both Findtool and Survey were developed by reusing (and modifying) existing software. A generic system design permitted the prototype, Findtool, and the final product, Survey, to be implemented as variants of each other. Reusing a generic system design ensured that knowledge and experience gained prototyping was transferred to Survey. The investment in prototyping was minimized by the large amount of software reused constructing the generic system and the prototype.

Prototyping built a stock of variants. Variants of generic services produced for Findtool, if not reused without change, were more easily converted for use in Survey because the modules were better understood through prototyping.

6.4 Future research

The development of the Rigi project at the University of Victoria continues. The

current version of Survey enhances the flexibility of the Rigi Editor. Despite the power of this scripting language, the needs of the project have already stretched it to the limit.

What efforts would yield the most improvement to Survey? An upgraded version of Survey would take on a LISP flavor where the atoms are Rigi objects. Tinkertoy is an example of a visual interface to the LISP programming language [Edel 88].

Survey could be designed based on a single datatype: sets of Rigi objects. Survey predicates would return these sets rather than simple boolean values. All Survey scripts would be functions that accept and return one of these sets as its parameter list. If none were specified, the selected set could be assumed.

A Survey script could be initiated from a "Start" item in its icon menu and the current selected set would be transferred to the script for processing. A Survey icon placed in the Survey script would be executed as a subroutine with either a locally declared set or a globally selected set as its parameter.

Rigi objects would not be automatically delivered to Survey. The objective of Survey would be to mimic a user operating the Rigi Editor: building sets; following dependencies; linking and unlinking objects; using the clipboard to cut, copy and paste; processing the contents of various objects; and so on.

If a generic system design is maintained, this revision of Survey would provide more opportunities for software reusability. The technique of creating variants to generic system services would continue to produce dividends because modifications are made to well-understood software.

Bibliography

[AKW 79] Aho, A. V., Kernighan, B.W., Weinberger, P. J., *"Awk — A Pattern Scanning and Processing Language,"* AT&T Bell Laboratories, 1979.

[Bach 73] Bachman, C. W., "The Programmer as Navigator," *Comm. of the ACM*, November 1973.

[BoDD 85] Boehm, H., Demers, A., Donahue, J., *"A Programmer's Introduction to Russell,"* Tech. Report TR85-16, Computer Science Dept., Cornell University, 1985.

[BrMS 84] Brodie, M., Mylopoulos J., Schmidt J. (eds.), *On Conceptual Modelling*, Springer-Verlag, 1984.

[CTYY 89] Chang, S.-K., Tauber, M.J., Yu, B., Yu, J.-S., "A Visual Language Compiler," *IEEE Transactions on Software Engineering*, Vol. SE-15, No. 5, pp. 506-525, May 1989.

[Dona 85] Donahue, J., "Integration Mechanisms in Cedar," *Proc. of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, Seattle June 1985, *ACM SIGPLAN Notices*, Vol. 20, No. 7, pp. 245-251, July 1985.

[DMNy 70] Dahl, O-J., Myrhaug, B., Nygaard, K., *"SIMULA Common Base Language,"* Norwegian Computing Centre S-22, Oslo, Norway, 1970.

[Edel 88] Edel, M., "The Tinkertoy Graphical Programming Environment," *IEEE Transactions on Software Engineering*, Vol. SE-14, No. 8, pp. 1110-1115, August 1988.

[Feld 79] Feldman, S., I., *"Make — A Program for Maintaining Computer Programs,"* Software - Practice & Experience, Vol. 9, No. 3, pp. 255-265, March 1979.

[FiGo 84] Finzer, W., Gould, L., "Programming by Rehearsal," Byte, Vol. 9, No. 6, pp. 187-210, June 1984.

[Free 83] Freeman, P., "Reusable Software Engineering Concepts and Research Directions", *Proc. of the Workshop on Reusability in Programming*, Newport, pp. 2-16, September 1983.

[Glin 86] Glinert, E. P., "Towards 'Second Generation' Interactive Graphical Programming Environments," *Proc. IEEE Computer Society Workshop on Visual Languages*, Dallas, pp. 61-70, June 1986.

[GlTa 84] Glinert, E. P., Tanimoto, S. L., "Pict: An Interactive Graphical Programming Environment," *IEEE Computer*, Vol. 17, No. 11, pp. 7-25, November 1984.

[GoRo 83] Goldberg, A., Robson, D., "*Smalltalk-80: The Language and its Implementation*", Addison-Wesley, 1983.

[Ichb 79] Ichbiah, J. D., *et al.*, "Rational for the Design of the ADA Programming Language," *ACM SIGPLAN Notices*, Vol. 14, No. 6, May 1979.

[Klas 88] Klashinsky, K., "A Practical Implementation of an Environment for Programming-in-the-Large," M.Sc. Thesis, University of Victoria, Victoria, B. C., 1988.

[Kern 84] Kernighan, B. W., "The UNIX System and Software Reusability," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, pp. 513-518, September 1984.

[MHHL 87] Müller, H. A., Hoffman, D. M., Horspool, R. N., Levy, M. R., "K2 — A Software Development Environment for Programming-in-the-large," *Proc. of the Conference on Intelligence Integration, CIPS Edmonton '87*, pp. 62-68, November 1987.

[MüKl 88] Müller, H., Klashinsky K., "Rigi — A System for Programming-in-the-large," *Proc. 10th International Conference on Software Engineering*, Singapore, pp. 80-86, April 1988.

[Müll 86] Müller, H., "Rigi — A Model for Software System Construction, Integration, and Evolution based on Module Interface Specifications," Ph.D. Thesis, Rice University, Houston, Texas, COMP TR86-36, 1986.

[NaSh 73] Nassi, I., Shneiderman, B., "Flowchart Techniques for Structured Programming," *ACM Sigplan Notices*, Vol. 8, No. 8, pp. 12-26, August 1973.

[Nels 80] Nelson, T., "Replacing the Printed Word: A Complete Literary System," *Proc. IFIP,* 1980.

[PaCW 83] Parnas, D. L., Clements, P. C., Weiss, D. M., "Enhancing Reusability with Information Hiding," *Proc. of the Workshop on Reusability in Programming*, Newport, pp. 240-247, September 1983.

[Parn 72] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," *Comm. of the ACM*, Vol. 15, No. 12, December 1972.

[PoNg 83] Pong, M. C., Ng, N., "PIGS — A System for Programming with Interactive Graphical Support," *Software — Practice and Experience*, Vol. 13, No. 9, pp. 847-855, September 1983.

[Reis 85] Reiss, S. P., "PECAN: Program Development Systems that Support Multiple Views," *IEEE Transactions on Software Engineering*, Vol. SE-11, Vol. 3, pp. 276-284, March 1985.

[Shaw 84] Shaw, M., "Abstraction Techniques in Modern Programming Languages," *IEEE Software*, Vol. 1, No. 4, pp. 10-27, October 1984.

[SPSS 83] *SPSS^X User's Guide*, SPSS, Inc., McGraw-Hill, 1983.

[Stro 86] Stroustrup, B., "*The C++ Programming Language*," Addison-Wesley, 1986.

[SUN 88a] *SunOS Reference Manual*, Sun Microsystems, Inc., May, 1988.

[SUN 88b] *SunView 1 Programmer's Guide*, Sun Microsystems, Inc., May, 1988.

[TaGl 86] Tanimoto, S. L., Glinert, E. P., "Designing Iconic Programming Systems: Representation and Learnability," *Proc. of the IEEE Computer Society Workshop on Visual Languages*, Dallas, pp. 54-60, June 1986.

[Wegn 84] Wegner, P., "Capital-Intensive Software Technology," *IEEE Software*, Vol. 1, No. 3, pp. 7-45, July 1984

[Wino 84] Winograd, T., "Beyond Programming Languages", *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall (eds.), pp. 517-534, McGraw-Hill, 1984.

[Wirt 83] Wirth, N., *Programming in Modula-2*, Second Edition, Springer-Verlag, 1983.

[Wood 81] Woods, J. A., *Find* (source code), Informatics General Corp., NASA Ames Research Center, June 1981.

[Zloo 80] Zloof, M. M., "A Language for Office and Business Automation," *1980 Office Automation Conf. Digest*, AFIPS Press, pp. 249-260, 1980.

Appendix A
Find

The UNIX file system is an hierarchical tree of files and directories. Directories are special files providing access to still further files and directories. Links can destroy the directory hierarchy, but they shall be ignored for the purpose of this discussion. The organization of files and directories can represent relationships.

If a large software project is being developed using the UNIX file system, there are hundreds or even thousands of files in layer upon layer of directories. Probably no single project team member is completely aware of the structure or contents of the project database. To assist in file management tasks the *find* command exploits the hierarchy of the UNIX file system.

```
find  pathname-list  expression
```

Find performs an operation on all files and directories *rooted at* (descended from) a directory in the pathname-list. Links are ignored to avoid inadvertent infinite loops. *Find* predicates test file attributes maintained by the file system. The contents of files are processed by invoking other system commands.

The utility of *find* is best demonstrated by example. Suppose all object files associated with a given project are to be removed. Let us assume that the root directory containing the project's files is called `project1` and that, by convention, all object files have the suffix `.o`. The project librarian could commence this maintenance by

listing the contents of each and every directory and issuing commands to remove each file with a name ending with `.o`. A simple *find* script could perform the same work automatically, with faster turnaround, and be less tedious and error prone.

```
find project1 -name '*.o' -exec rm {} \;
```

This *find* script recursively descents from the `project1` directory, invoking the system command to remove each file or directory whose name matches the regular expression `*.o` (meaning any filename ending with `.o`). Regular expressions and other strings of interest to the system *shell* must be "protected" from its filename expansion features to ensure that the expression is submitted correctly to *find*. Quoting of strings and escape characters temporarily disables filename expansion.

Find's file-attribute predicates are described briefly here. Full documentation on *find* is contained in the SUN OS Commands Reference Manual [SUN 86a].

-name filename match filename

The filename argument is parameterized using standard *shell* metacharacters:

'*' - any character string

'?' - any single character

'[]' - any of the enclosed characters

(ranges are specified as a pair of characters separated by '-')

-perm onum validate file permission (refer to *chmod* for onum codes)

-type c check filetype (b - block special file, c - character special file,

	d- directory, f - file, p - pipe, l - symbolic link, s - socket)
-user uname	file owned by uname (login or userid)
-nouser	file is not owned by any known user
-group gname	file belongs to group gname (name or group id)
-nogroup	file's group not recognized
-newer file	true if current file modified after argument file
-links n	number of links
-size n	size of file (in 512 byte blocks, characters if followed by 'c')
-inum n	inode number
-atime n	days since last access
-mtime n	days since last modification
-ctime n	days since last change (modification of contents or attributes)

The argument 'n' represents a decimal integer where '+n' means "greater than n," '-n' "less than n," and 'n' "exactly n."

Find also supports predicates with side-effects. Assume these predicates return true trivially unless specifically stated otherwise:

-print	print pathname
-ls	display /bin/ls -gilds (ls is done independently)
-prune	discontinue descent here
-xdev	prune if about to descent into a different file system
-depth	search file system depth-first (default: breadth-first)

-cpio device	dump file to device in cpio format
-ncpio device	dump file to device in cpio -c format
-exec command	invoke system command
-ok command	if user responds 'y' to prompt invoke system command

The versatility of *find* lies in the application of the `-exec` predicate. All text up to and including a single (escaped) ';' is passed to the system command interpreter for execution. The current filename is substituted for the string '{}'. If the command terminates with return code 0, the `-exec` predicate returns true, otherwise false.

Three additional boolean operators combine predicates into an expression:

expr **-a** expr	logical and of expressions
expr expr	-a operator is 'understood'
expr **-o** expr	logical or of expressions
! expr	logical not of expression

The left-to-right precedence rule may be circumvented by parentheses. Short-circuit evaluation is implemented for both the `-a` and `-o` operators: the second expression is only evaluated if it could affect the final result.

Appendix B
Running Findtool

An account environment must be prepared to use Findtool. The directory where the system manager has installed the special files for Findtool's support is made accessible via the `FINDTOOLDIR` environment variable. Supposing the Findtool support directory was located at images/findtool, then include

```
setenv  FINDTOOLDIR  images/findtool
```

Where is the Findtool expression? Findtool maintains expressions in UNIX directories. Findtool expects the user to have read/write privileges on this directory. The pathname to the directory is supplied using the UNIX environment variable `FINDDIR`. To instruct Findtool to refer to the expression maintained in the directory `findme` use the following

```
setenv  FINDDIR  findme
```

If a directory called `findme` cannot be located, a new one is allocated. If the user prefers not to use `FINDDIR`, an expression in `.find` is sought.

Once a path to the Findtool executable is located, Findtool is started by typing

```
findtool   pathname-list
```

where the pathnames appear as they would for *find*. Unlike *find*, if the pathname-list is omitted Findtool understands that the current directory symbol '.' is implied.

When Findtool is started two SunView windows are displayed. The first,

labelled "*Findtool V1.0*," is a drawing "canvas" for the Findtool Editor. The other is a non-editable text window for Findtool output. The window layout is depicted in Figure 3.1. The user has control under suntools to re-position, re-size, or scroll through the window contents. The "*Done*" frame menu item hides the output window. It may be exposed by selecting "*Props*" from the edit window's frame menu.

All Findtool predicates are listed under the *Findtool Predicates* menu item in the background menu. When a predicate is selected, a new icon corresponding to the predicate appears on the drawing "canvas". Since some of the icons may be obscure at first, the predicates are listed in the menu under their *find* command keywords.

A predicate is selected by left-clicking its icon. The shift modifier key allows predicates to be added incrementally to the selection. A predicate may be removed from the selected set by shift-left-clicking it. As in the Macintosh user interface, a selection may also be made by left-dragging an enclosing rectangle originating at a point in the background and encompassing all desired predicates in the bounding box. The selected predicates constitute the selected set and are highlighted. Most editing operations apply to the selected set. A new selection releases the previously selected set.

With *find*, evaluation precedence is governed by parentheses and a left-to-right scanning rule. Parentheses are not required within Findtool because the layout of the abstract syntax tree governs evaluation precedence. However, there is still an implicit left-to-right ordering of predicates.

In the abstract syntax tree, arcs are drawn from the bottom of a parent to its

children. A connecting arc is inserted from each member of the selected set to a middle-clicked predicate. Or, if the shift modifier key is held down, the arcs are inserted from that predicate to each member of the selected set. To remove an arc, insert it a second time.

The predicates may be re-arranged on the "canvas" by middle-dragging them to their desired positions. If the shift modifier key is applied, the entire selected set moves in unison. As a predicate is dragged about, its connecting arcs are re-drawn interactively with an elastic rubber-band effect.

Findtool has imported from the Rigi Editor the Macintosh notion of cut/copy/paste editing. These editing primitives are supported by a hidden data store: the *clipboard*. Findtool predicates can be moved temporarily to the clipboard for processing. Use the "*Show Clipboard*" menu item to view and edit the contents of the clipboard.

Copying creates an exact copy of the selected predicates on the clipboard. When copying the selected set, any arcs to or from predicates not included in the set are ignored. Arcs within the selected set are copied intact. A *cut* performs a copy, but it also deletes the selected predicates from the Findtool expression. Any arcs to or from the deleted predicates, copied to the clipboard or not, are deleted. A cut or copy overwrites the previous clipboard contents.

When the clipboard contains an expression, a copy of it may be *pasted* into the Findtool expression. The predicates just pasted become the new selected set. Cutting and pasting is memoryless with respect to arcs: the arcs lost during the cut must be restored manually. The clipboard may be duplicated as many times as desired; each

copy of the expression pasted is unique.

Findtool predicates accept the same parameters as their *find* "cousins." Findtool does not have to contend with filename expansion because Findtool expressions are not received through the *shell*. Therefore, all the contortions of syntax learned to cope with *find* are unnecessary. Parameter entry mode is entered through the predicate's icon menu. If the predicate does not permit parameters, the menu selection is disabled. Like the predicates of *find*, Findtool predicates are categorized by the number of parameters they expect:

1. Predicates with no parameters;
2. Predicates with a single one-word parameter;
3. Predicates which accept an arbitrary string parameter.

If the predicate belongs to the single one-word category, all keyboard characters typed are appended to the parameter field until terminated by a carriage return or any mouse event. When a new parameter is entered, the previous one is discarded.

Otherwise, the predicate can be expected to contain too much text to display conveniently beneath the predicate's icon. Thus, a standard SunView text editor pop-up window, as depicted in Figure 3.1, is used to edit the parameter text. All events are processed by this pop-up while editing the parameter text. This parameter text may be retrieved repeatedly for editing. Even though the *shell* syntax is a thing of the past, the `-exec` parameters still need to be terminated by a semicolon. The first line of text is displayed in the icon's name field like a single one-word parameter.

The Findtool expression is executed by "*Find*" from the background menu of the Findtool Editor.

Incorrect syntax causes *find* to print error messages; Findtool uses the same error messages but displays the error condition in a pop-up dialog box. The error message continues to be displayed until acknowledged by a left-click and then control returns to the Findtool Editor. The Findtool expression may be edited and re-run. A complete listing of Findtool error messages is included below:

```
findtool: can't chdir back to ...
findtool: lost home ...
findtool: bad start directory
findtool: newer cannot access ...
findtool: cannot stat ...
findtool: pathname too long.
findtool: cannot chdir to ...
findtool: cannot open ...
findtool: bad directory tree (cannot chdir to ... )
findtool: couldn't find mount point for ...
findtool: syntax error at ...
findtool: parsing error
findtool: missing conjunction
```

```
findtool: AND must have at least TWO operands
findtool: OR must have at least TWO operands
findtool: -exec missing ';'
findtool: -ok missing ';'
findtool: cannot find -user name ...
findtool: cannot find -group name ...
findtool: cannot create ... for cpio
findtool: too many options
findtool: incomplete expression
findtool: bad malloc
```

Findtool is exited like any other SunView application, i.e., choosing "*Quit*" from the Findtool Edit Window's frame menu. If quitting is confirmed, Findtool closes its windows. Before actually quitting, however, the contents of the current database are saved to `FINDDIR` automatically. This Findtool expression may be run or edited the next time it is specified as the `FINDDIR`.

Appendix C
Survey Error Messages

```
survey: can't locate graph ...

survey: lost home ...

survey: bad start graph

survey: no expression found

survey: multiple expressions selected

survey: cannot open ...

survey: syntax error at ...

survey: parsing error

survey: missing conjunction

survey: AND must have at least TWO operands

survey: OR must have at least TWO operands

survey: too many options

survey: incomplete expression

survey: bad malloc

survey: invalid node type

survey: invalid arc type

survey: unknown document type
```

VITA

Surname: **Idler** Given Names: **Ernest Alan**
Place of Birth: **New Westminster, B. C.** Date of Birth: **May 16, 1959**

Educational Institutions Attended, with Dates of Entering and Leaving:

University of British Columbia	1977 to 1981
University of Victoria, B.C.	1986 to 1989

Degrees, Diplomas, Etc., Awarded, with Dates and Names of Institutions:

B.Sc.	1981	University of British Columbia

Partial Copyright License

I hereby grant the right to lend my thesis (the title of which is shown below) to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

A Visual Scripting Language

Author: [redacted]

Ernest Alan Idler

July 31, 1989

ISBN 0-315-53761-2