Domain–Retargetable Reverse Engineering II: Personalized User Interfaces†

Scott R. Tilley

Department of Computer Science, University of Victoria
P.O. Box 3055, Victoria, BC, Canada V8W 3P6
Tel: (604) 721-7294, Fax: (604) 721-7292
E-mail: stilley@csr.uvic.ca

Abstract

The user interface is an integral part of any application. This is especially true for reverse engineering environments, since the understanding process depends both on the user's specific cognitive abilities and on the palette of tools accessible through the interface. Because software engineers approach program understanding in so many different ways, it is impossible to predict what they want from the supporting environment in all situations. Therefore, the user interface should be malleable and customizable. By having such a personalizable *user interface, users can adapt the environment to their particular needs and "taste," while still maintaining a common "look and feel."*

Keywords: domain-retargetable, program understanding, user interfaces

1 Introduction

While much work has been done on modeling how software engineers understand programs, no single environment will ever be suitable for all users in all application scenarios. The disparate nature of users' cognitive abilities and their diverse approaches to program understanding preclude the use of a static suite of selection, analysis, and structuring techniques. Similarly, the user interface to the toolset provided by the supporting environment cannot be inflexible. Because the interface provides access to the tools and shapes the users' view of the underlying environment, it too must be dynamic and customizable.

†This work was supported in part by the British Columbia Advanced Systems Institute, the IBM Software Solutions Toronto Laboratory, the IRIS Federal Centres of Excellence, the Natural Sciences and Engineering Research Council of Canada, the Science Council of British Columbia, and the University of Victoria.

Unfortunately, the attitude that seems prevalent to many program understanding tool builders is that "if programmers would just learn to understand code the way they ought to" (i.e., the way the builders' tools work), the code comprehension problem would be solved [1]. It is now apparent that such a builder-oriented view is doomed to failure for the analysis of large programs. Instead, we should provide user interfaces that support the users' view; interfaces that *support* the natural process of program understanding—not hinder it.

Perhaps the most important goal for a successful program understanding environment is to provide a mechanism through which users can extend the system's functionality. For example, it should be possible to augment the search and selection operations with user-defined algorithms and external tools. The environment should give users as much freedom as possible in configuring the system to their liking. This configurability includes modification of the system's interface components such as buttons, dialogs, menus, scrollbars, and so on. Experienced users should be able to create time-saving meta-commands or "accelerator" key sequences. More importantly, for the environment to be applicable to multiple domains, it should be possible to alter the system's functionality by changing the commands associated with elements of the user interface.

Since the user interface is a crucial part of the infrastructure of many software environments [2], particularly environments for program understanding, and since personal preferences for such things as menu structure, mouse action, and system functionality differ so much from person to person (and from domain to domain), it is unlikely that any single choice made by the tool builder will suit all users. Ultimately, it is the user—not the builder—who decides if the user interface of an application is adequate.

This paper describes on-going work on *domain-retargetable reverse engineering*, as discussed in [3].

Previous efforts were directed towards extending the functionality of an existing environment via *programmable reverse engineering* [4], a mechanism that enables users to write their own routines for common reverse activities such as extraction, selection, analysis, and organization of information artifacts. This paper's focus is on Phase II: making the environment's user interface tailorable. Specifically, it advocates *personalized* user interfaces for program understanding environments.

The next section discusses three issues of importance to program understanding and describes how they are affected by deficiencies in traditional user interfaces. Section 3 describes an approach to personalized user interfaces in a reverse engineering environment for program understanding. The approach is based on end-user customization of the environment's user interface through a scripting language. Section 4 outlines how personalized user interfaces are used and gives several examples of their application. Section 5 summarizes the paper and briefly discusses future work.

2 Deficiencies in traditional user interfaces

Programmers make use of programming knowledge, domain knowledge, and comprehension strategies when attempting to understand a program. One extracts syntactic information from the source code, and relies on programming knowledge to form semantic abstractions. Brooks's work on the theory of domain bridging [5] describes the programming process as one of constructing mappings from a problem domain to an implementation domain, possibly through multiple levels. Program understanding, then, involves reconstructing part or all of these mappings. The process is expectation-driven, and proceeds by creation, confirmation, and refinement of hypothesis. It requires both intra- and inter-domain knowledge. A problem with this reverse mapping approach is that mapping from application to implementation is one-to-many, because there are many ways of implementing a concept.

While there are several deficiencies in traditional user interfaces, the lack of programmability particularly affects program understanding environments in three ways: cognitive models, domain retargetability, and tool integration. Each of these is discussed in more detail below.

2.1 Cognitive models

It is difficult for any application designer to predict all the ways an application will be used. In a program understanding environment, the main goal is to facilitate code comprehension. Because people learn in different ways—for for example, goal-directed (top-down and inductive) versus scavenging (bottom-up and deductive)—the environment should be flexible enough to support different types of comprehension.

Two common approaches to code comprehension often cited in the literature are a functional approach that emphasizes cognition by *what* the program does, and a control-flow approach that emphasizes *how* the program works. Both top-down and bottom-up comprehension models have been used in an attempt to define how a software engineer understands a program. However, case studies have shown that maintainers frequently switch between several comprehension strategies [6]. Therefore, the tools and environment must support the diverse cognitive processes of program understanding, rather than impose a process that is not justified by a cognitive model other than that of the environment's developers.

Presentation integration can occur at different levels, including the window system, the window manager, the toolkit used to build applications, and the toolkit's *look and feel* [7]. The standardization provided by presentation integration lessens the "cognitive surprise" experienced by users when switching between tools. However, what is really needed is a way for the user to specify the common look and feel of the applications of interest to them, or of tools that are part of an application. In other words, users need to be able to impose their own personal *taste* on the common look and feel. This refinement of presentation integration moves the onus—and the opportunity—for reducing cognitive overhead due to the user interface from the tool builder to the tool user.

2.2 Domain retargetability

One way of maximizing the usefulness of a program understanding system is to make it domain-specific. By doing so, one can provide users with a system tailored to a certain task and exploit any features that make performing this task easier. However, by taking this approach the system's usefulness becomes restricted to a particular domain. Using the same system on a different task, even one that is similar, may be impossible.

An alternative to making the system powerful by making it domain-specific, is to make it user-programmable and hence domain-retargetable. One would like to make the approach as flexible as possible—a subtle distinction from making it general. Software can be considered *general* if it can be used without change; it is *flexible* if it can be easily adapted to be used in a variety of situations [8]. General solutions often suffer from poor performance or lack of features that limit their usefulness. Flexible solutions may be tailored by the user to fully exploit aspects of the problem that make its solution easier.

Flexibility and *scalability* are key requirements for a successful program understanding environment. The approach must be flexible so that the results can be applied to many diverse program understanding scenarios as well as different target domains. "Domains" in this sense is an over-burdened term. It refers to different *application* domains, such as banking or database systems; *implementation* domains, including the application's implementation language; and the *reverse engineering* domain, in which the software engineer models and represents the subject system. The approach must also be applicable to large software systems, on the order of several million lines of code. Such a scale precludes the use of programming-in-the-small approaches to program understanding.

2.3 Tool integration

Most existing reverse engineering systems provide the user with a fixed set of view mechanisms, such as call graphs and module charts. While this set might be considered large by the system's producers, there will always be users who want something else. One cannot predict which aspects of a system are important for all users, and how these aspects should be documented, represented, and presented to the user. This is an example of the trade-off between open and closed systems. An open system provides a few primitive operations and mechanisms for user-defined extensions. A closed system provides a "large" set of built-in facilities, but no way of extending the set.

There are two basic approaches to constructing integrated applications from a set of tools: tool *integration* and tool *composition* [9]. In tool integration, each tool must be aware of the larger environment, and the inter-tool interactions are coded in the tools themselves. This works for tightly-integrated environments, but in a loosely-coupled environment it is very difficult to achieve. In tool composition, tool interaction logic resides outside of the tools. Each tool presents a standard, well-known interface to the outside world, and knows nothing about its environment; the environment contains all the inter-tool coordination logic. From an end-user perspective, the program understanding environment should manage tool composition, to facilitate the introduction of new tools into the system. This includes extending the user interface with new widgets, additional tools, and domain-specific *personalities*.

The program understanding process is extremely dependent on personal taste and approaches to the problem. Therefore, the supporting environment requires the integration of different technologies, applications, and analysis tools. One way of achieving this goal is through control and presentation integration. The user would then be able to alternate and select among these tools as required. Communication in such a distributed environment can be achieved by scripts that each tool understands. Such an approach is described in the next section.

3 Supporting personalized user interfaces

The success of moving new technology into the workplace depends crucially on the acceptance of the system by its users. If they find the system too different from what they are currently using, they will be loath to change. However, if they can *tailor* the new system to fit into their existing environment and work the way they want it to—not the way the designer thought they might want it to work—then the system has a much better chance of success.

Hence, the goal is to provide the users with as much flexibility as possible in structuring the environment to suit their needs. Such extensibility has proven effective in other domains, such as hypertext systems [10], custom database applications, and Computer-Aided Design (CAD) systems. Business application suites are also beginning to provide scripting languages as a kind of unifying coordination mechanism. For example, Microsoft[1] intends to use Visual Basic for Applications (VBA) [11] as a common extension language for its suite of office applications. Similarly, Lotus is planning a cross-application scripting language for its product offerings. The customizable user interface of WordPerfect[2] Six.0 is extolled as a significant value-added feature. In these suites, users have the power to tailor and configure menu bars, status bars, style ribbons, and scroll bars.

To meet the goal of personalized user interfaces, the Rigi system has been extended. Rigi is a versatile system and framework under development at the

[1]Trademark of Microsoft Corporation.

[2]Trademark of WordPerfect Corporation.

University of Victoria for analyzing the structure of large software systems. It consists of a variety of parsing systems supporting the extraction of information from source code or documentation, a repository to store the information extracted, and rigiedit: an interactive graph editor that permits graphical manipulation of the underlying conceptual structures. A more detailed description of Rigi and its approach to reverse engineering can be found in [12].

As outlined in [3], in 1993 we embarked on a two-phased extension to rigiedit to make the environment domain retargetable. The first phase was to extend the editor's functionality through the inclusion of a scripting language. The second phase was to make the user interface tailorable. This section briefly outlines the realization of the second phase: enabling personalized user interfaces. This was done through the use of the scripting language previously incorporated to extend system functionality, and the use of an interface toolkit that allowed us to incrementally retrofit a new user interface onto an existing one.

3.1 Phase I: Incorporating a scripting language

Phase I of the changes to rigiedit concentrated on the decoupling of the graph editor from the graphical user interface. A transparent scripting layer was then introduced between the direct-manipulation user interface and the graph editor. This made it possible to program editor operations independently of graphical user interactions.

A scripting language amplifies the power of the environment by allowing users to write scripts to extend the tool's facilities. Examples of successful tools that are end-user programmable include spreadsheets [13] and the UNIX[3] shell. A scripting language is extremely useful in a windowing environment; it just works "behind the scenes". Users who accept the default will be unaware of a scripting language—but its power is available to those who want to take advantage of it.

Rather than write yet another command language, the Rigi command language RCL was built on top of Tcl [14]. Tcl provides an extendable core language, and was specifically written as a command language for interactive windowing applications. It also provides a convenient framework for communicating between Tcl-based tools. Each application extends the Tcl core by implementing new commands that are indistinguishable from built-in commands, but are specific to the application. Tcl is application-independent and provides two different interfaces: a textual interface to users who issue Tcl commands, and a procedural interface to the application in which it is embedded.

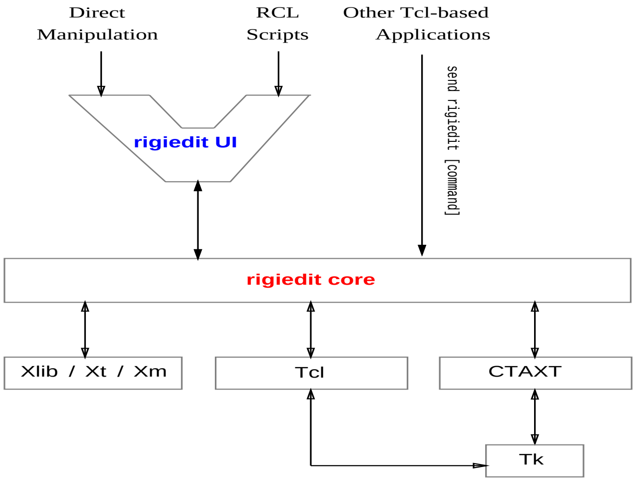

Figure 1: Extending `rigiedit`

3.2 Phase II: Personalizing the user interface

Besides using Tcl as a scripting language to enable users to configure the program understanding environment's actions, one may use it in conjunction with the Tk toolkit to configure the environment's interface. Tk is an X11 toolkit that implements the Motif[4] look and feel. It is similar in functionality to the Xm toolkit which we previously used for our Motif interface, except that it may be programmed in Tcl.

Rather than having to redo the entire user interface from scratch, the rigiedit architecture was incrementally augmented the existing with Tk widgets. To do this, the CTAXT [15] interface library was used. CTAXT enables Tk widgets to coexist with Xm widgets in the same application. Without CTAXT, problems arise with allocation of window manager signals and resources. A pictorial representation of the new rigiedit architecture is shown in Figure 1. All commands, whether they originate from the direct manipulation graphical user interface or from RCL scripts, go through the same routing and logging mechanism in the rigiedit core. This includes remote command sequences as well.[5]

[3] UNIX is a registered trademark licensed exclusively by X/Open Company, Ltd.

[4] Motif is a trademark of Open Software Foundation.

[5] Tcl provides a `send` command to enable one applica-

By using Tcl and Tk as an intermediary for all interface actions, users can write Tcl scripts to personalize the layout and appearance of the environment as desired. The use of some of these tailoring facilities is discussed in the next section.

4 Using personalized user interfaces

The user interface is customized and extended in a manner very similar to how the functionality of the program understanding environment is customized and extended. Users can write scripts that are read automatically upon rigiedit invocation to tailor the interface. The same scripts can be used to dynamically alter the user interface by reconsulting the source file while rigiedit is running. Such *on-the-fly* alterations of the user interface is difficult using other methods, such as the OSF/Motif User Interface Language (UIL) [16].

The rigiedit user interface looks at three environment variables to determine its "taste." Beyond the standard X resources file Rigiedit, the environment variables RIGISTY and RIGIUSTY are used to represent system-defined and user-defined extensions to the core user interface functions, respectively. These files serve purposes similar to their RIGIRCL/RIGIURCL toolset counterparts: they are for programming the system default and user preferences of the user interface, respectively. A command-line option may also be used to load session preferences. Thus, the user interfaces preferences are applied in layers: Rigiedit, then RIGISTY, then RIGIUSTY, then command-line settings.

To illustrate the use of dynamic alteration of the user interface, a small code fragment used to tailor the Layout menu (used in the example below) is shown in Figure 2. The RCL code shown in the Figure is more complicated than that which would normally be used; adding a menu item to an existing menu can be accomplished through the use of higher-level parameterized procedures, or through the use of interface builders for Tk, such as XF [17].

To illustrate the use of the personalized user interface as discussed above and in Section 3, the analysis of a sample C program is used. One method of reverse engineering C programs is to group data types and their access functions into logical components (subsystems). The original programmer may or may not have done this type of abstract datatype (ADT) clustering

tion to control another across the network.

```
# Personalized layout menu [SRT 22Mar94]

# Assumes .rigi.menubar.layout.m already
# exists from default RIGISTY file.

.rigi.menubar.layout.m add separator
.rigi.menubar.layout.m add command\
   -label "Spring"\
   -bitmap @/home/stilley/icons/spring.xbm\
   -command "layout gel-spring"\
   -underline 0\
   -accelerator Alt+S
pack .rigi.menubar.layout
```

Figure 2: Personalizing the layout menu

in the actual source code, but experience indicates it is a useful first step in C program understanding.

There are at least two ways of identifying ADT subsystem candidates: visually and analytically. Visual identification is sometimes possible through the use of an appropriate layout algorithm, one that clusters artifacts appropriately. A spring layout [18] does this, since it models the subject software system as a physical system, using node positions and arc strengths to simulate attraction and repulsion among nodes.

Figure 3 shows a sample rigiedit session. The Layout menu has been torn off and extended as described above. The user has selected a subset of nodes in the C program and initiated two layouts: the first (shown in the left window of the figure) reflects the control flow of the program, using a builtin tree layout algorithm [19]; the second (shown in the right window of the figure) reflects the data usage of the program using the off-line spring layout algorithm. Two nodes in the data-oriented window clearly stand out as "interesting": *list* and *element*. These are C `struct` data types. This visual information could be used to guide the user towards a decomposition based on data encapsulation.

Other examples of tailoring the user interface to aid program understanding include using external tools (such as a spreadsheet) to visualize, manipulate, and maintain interface dependency information produced by the editor, attaching operations to be invoked when subsystem nodes are selected and "opened" (such as indicating the number of nodes and arcs in the subsystem), and even replacing the default canvas with another presentation tool (such as a more advanced animation and/or visualization system). These types

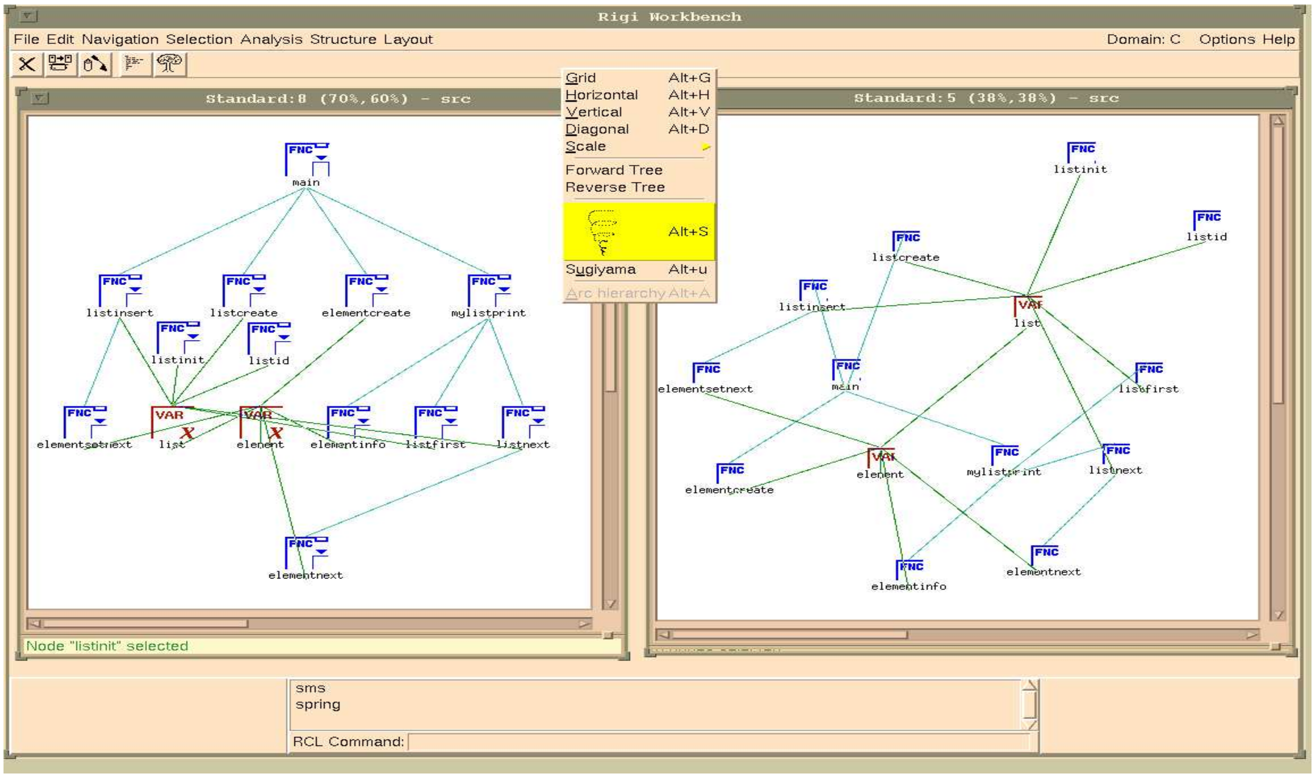

Figure 3: Visual identification of abstract data type

of customization are easy for the end-user to specify; they enable integration of foreign tools with the graph editor; and they allow users to leverage their human cognitive abilities in deciding what is the best method of information presentation for their current application domain.

5 Summary

The understanding process is dependent on both individuals and their specific cognitive abilities, and on the (sometimes limited) set of facilities that tools provide. Understanding also requires the ability to adapt to various application domains, implementation languages, and working environments. The approach taken is based on customizable user interfaces, which give individuals the ability to tailor their environment to suit their needs.

The approach achieves flexibility partially through the incorporation of a scripting language that provides an interfacing mechanism. The scripts give users the ability to extend the tools in their reverse engineering toolbox by defining, storing, and retrieving commonly used operations. Using Tcl as the scripting language and Tk as the user interface toolkit enabled us to leverage skills of other people and groups who write Tcl scripts. If we had written our own scripting language, this would not have been possible to nearly the same extent.

Rather than provide a single interface to a general toolset, the persistent problem of "what is a good user interface for this application" is solved—not by the tool builder, but by the tool user. By providing a user-programmable program understanding environment, the application domain need not be limited to one area. While most program understanding environments provide a fixed palette of analysis, selection, and presentation techniques, our approach uses a scripting language that enables users to write their own routines for these activities. Users can adapt the environment to suit their own personal *taste*, while still working within the common *look and feel* imposed by the window manager.

Future work will include the further investigation of the integration of rigiedit with other reverse engineering tools. Preliminary results indicate that an extensible but integrated toolkit is required to support the multi-faceted analysis necessary to understand legacy software systems.

Acknowledgments

The support work of Brian Corrie and Michael Whitney is greatly appreciated.

References

[1] A. von Mayrhauser and A. M. Vans. From code understanding needs to reverse engineering tools capabilities. In *CASE '93: The Sixth International Conference on Computer-Aided Software Engineering,* (Institute of Systems Science, National University of Singapore, Singapore; July 19-23, 1993), pages 230–239, July 1993. IEEE Computer Society Press (Order Number 3480-02).

[2] M. Young, R. N. Taylor, and D. B. Troup. Software environment architectures and user interface facilities. *IEEE Transactions on Software Engineering*, 14(6):697–708, June 1988.

[3] S. R. Tilley, H. A. Müller, M. J. Whitney, and K. Wong. Domain-retargetable reverse engineering. In *CSM '93: The 1993 International Conference on Software Maintenance,* (Montréal, Québec; September 27-30, 1993), pages 142–151, September 1993. IEEE Computer Society Press (Order Number 4600-02).

[4] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller. Programmable reverse engineering. Submitted to the *International Journal of Software Engineering and Knowledge Enginering,* July 1994.

[5] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[6] E. M. Gellenbeck and C. R. Cook. An investigation of procedure and variable names as beacons during program comprehension. Technical Report 91-60-2, Computer Science Department, Oregon State University, 1991.

[7] A. I. Wasserman. Tool integration in software engineering environments. In G. Goos and J. Hartmanis, editors, *Proceedings of the International Workshop on Environments,* (Chinon, France; September 18-20, 1989), pages 137–149. Springer-Verlag, 1989.

[8] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–137, March 1979.

[9] A. K. Arora, D. W. Hurst, and J. C. Ferrans. Building diverse environments with PCTE workbench. In *PCTE '93*, 1993.

[10] P. D. Stotts and R. Furuta. Dynamic adaptation of hypertext structure. In *Proceedings of Hypertext '91* (San Antonio, Texas; December 15-18, 1991), pages 219–231, December 1991. ACM Order Number 614910.

[11] T. J. Biggerstaff. Directions in software development & maintenance. University of Victoria invited talk, December 9, 1993.

[12] H. Müller, S. Tilley, M. Orgun, B. Corrie, and N. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In *SIGSOFT '92: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments,* (Tyson's Corner, Virginia; December 9-11, 1992), pages 88–98, December 1992. In *ACM Software Engineering Notes*, 17(5).

[13] B. A. Myers. Why are human-computer interfaces difficult to implement? Technical Report CMU-CS-93-183, Computer Science Department, Carnegie Mellon University, July 1993.

[14] J. K. Ousterhout. *An Introduction to Tcl and Tk.* Addison-Wesley, 1994.

[15] D. Spruce and H. Pleiss. CTAXT – Combine Tcl/Tk with arbitrary X toolkits. Technical report, European Synchrotron Radiation Facility, January 1994.

[16] D. Heller and P. M. Ferguson. *Motif Programming Manual.* O'Reilly & Associates, Inc., 1994.

[17] S. Delmas. XF: Design and implementation of a programming environment for interactive construction of graphical user interfaces. Part of the *XF* distribution kit, Technische Universität Berlin, 1993.

[18] T. Fruchtermann and E. Reingold. Graph drawing by force-directed placement. Technical Report UIUC CDS-R-90-1609, Department of Computer Science, University of Urbana-Champaign, 1990.

[19] E. Reingold and J. Tilford. Tidier drawing of trees. *IEEE Transactions on Systems, Man, and Cybernetics*, SE-7(2), March 1981.