A WRAPPER GENERATION TOOL FOR THE CREATION OF SCRIPTABLE SCIENTIFIC APPLICATIONS

by

David M. Beazley

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

The University of Utah

August 1998

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

David M. Beazley

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

JUNE 4, 1998	[illegible] Chair: Christopher R. Johnson
6/4/98	[illegible] Alan Davis
July 31, 1998	[illegible] Gary Lindstrom
6/4/98	[illegible] John Carter
6/4/98	[illegible] Peter S. Lomdahl

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of David M. Beazley in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

JUNE 29, 1998
Date

Christopher R. Johnson
Chair, Supervisory Committee

Approved for the Major Department

Robert Kessler
Chair/Dean

Approved for the Graduate Council

Ann W. Hart
Dean of The Graduate School

ABSTRACT

In recent years, there has been considerable interest in the use of scripting languages as a mechanism for controlling and developing scientific software. Scripting languages allow scientific applications to be encapsulated in an interpreted environment similar to that found in commercial scientific packages such as MATLAB, Mathematica, and IDL. This improves the usability of scientific software by providing a powerful mechanism for specifying and controlling complex problems as well as giving users an interactive and exploratory problem solving environment. Scripting languages also provide a framework for building and integrating software components that allows tools be used in a more efficient manner. This streamlines the problem solving process and enables scientists to be more productive.

One of the most powerful features of modern scripting languages is their ability to be extended with code written in C, C++, or Fortran. This allows scientists to integrate existing scientific applications into a scripting language environment. Unfortunately, this integration is not easily accomplished due to the complexity of combining scripting languages with compiled code. To simplify the use of scripting languages, a compiler, SWIG (Simplified Wrapper and Interface Generator), has been developed. SWIG automates the construction of scripting language extension modules and allows existing programs written in C or C++ to be easily transformed into scriptable applications. This, in turn, improves the usability and organization of those programs.

The design and implementation of SWIG are described as well as strategies for building scriptable scientific applications. A detailed case study is presented in which SWIG has been used to transform a high performance molecular dynamics code at Los Alamos National Laboratory into a highly flexible scriptable application. This transformation revolutionized the use of this application and allowed scientists to perform large-scale materials simulations on an day-to-day basis. In addition, a user survey is presented in which SWIG is shown to greatly simplify the creation of scriptable applications, improve productivity, and enhance the usability of scientific programs.

For my parents.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This research would not be possible without the contributions and support of many people. First, I would like to thank all of the SWIG users who have provided bug reports, feedback, and suggestions for improvement. There are far too many people to thank individually, but you know who you are. Second, I would like to thank my collaborators Tim Germann, Brad Holian, Shujia Zhou, Ralf Makkula, Niels Jensen, and Wanshu Huang in the Theoretical Physics Division at Los Alamos National Laboratory. Paul Dubois and Brian Yang at Lawrence Livermore National Laboratory also provided many interesting discussions concerning the use of scripting languages and scientific applications. I would also like to acknowledge Chris Johnson and the Scientific Computing and Imaging group at the University of Utah for their generous support of this work. Finally, I like to offer a special thanks to Peter Lomdahl at Los Alamos National Laboratory who has supported my efforts throughout graduate school and allowed me to pursue crazy ideas.

This research has been performed under the auspices of the Department of Energy, National Science Foundation, and a University of Utah Graduate Research Fellowship.

CHAPTER 1

INTRODUCTION

Scripting languages such as Perl, Python, and Tcl are becoming an increasingly popular tool for the development and use of modern software. In fact, John Ousterhout, creator Tcl, writes:

> For the past 15 years, a fundamental change has been occurring in the way people write computer programs. The change is a transition from system programming languages such as C or C++ to scripting languages such as Perl or Tcl. Although many people are participating in the change, few realize that the change is occurring and ever fewer know why it is happening [77, p. 23].

Although scripting languages have been used in a variety of computing applications, this dissertation primarily focuses on the use of scripting languages with scientific software. A tool, SWIG, has been developed to simplify the integration of scripting languages with existing software written in C and C++. Furthermore, the use of SWIG and scripting languages are shown to have a tremendous impact on the development, organization, and use of scientific software.

Traditionally, scientific computing has been ignored by most of the computer science and software engineering community. Likewise, computational scientists often give little attention to modern software practice. This dissertation illustrates the practical application and impact of many modern software construction techniques including the use of scripting languages, software components, design patterns, software re-engineering, and interface building tools on scientific programs. While the emphasis is on scientific applications, many of the techniques and results presented are applicable to other areas of software development.

1.1 The Problems Facing Computational Scientists

Computational scientists have recently witnessed an unprecedented change in the environment in which scientific simulations are performed. This change has been fueled by a number of developments including huge increases in simulation sizes due to increased computing power, a shift in the types of scientific simulations being performed, and a variety of new software development techniques such as object-oriented programming and component frameworks.

Unfortunately, these developments have greatly increased the complexity of developing and performing scientific computations. This complexity manifests itself in a number of ways. For example, the fact that a scientific program might run on workstations, shared memory multiprocessors, distributed memory parallel computers, and clusters greatly complicates software development and has led some researchers to call for better language, software development, debugging, and tool support [78]. Large-scale simulations have resulted in large amounts of data that overwhelm existing hardware and software–a problem often referred to as the "data glut" [20]. The increased interest in complex unstructured three-dimensional simulations has created a need for new data analysis and visualization tools. When combined with the data-glut, researchers often talk about "visual supercomputing" and the construction of highly interactive data analysis systems [80, 35]. Even though each of these problems is unique, they are all symptoms of the increasingly complex nature of scientific computing and the breakdown of traditional approaches.

Although there are many facets to the complexity puzzle, one of the biggest problems facing computational scientists is the process by which scientific software is developed, assembled, and controlled. Not only is the development of new software more complicated, but scientists must work with a wide variety of existing packages, libraries, and tools. These components are often written in different languages, use a variety of programming styles, and make different assumptions about data layout, file formats, and user interfaces. As a result, many computational scientists find themselves spending a large amount of time fighting with a "witches brew" of different programs, tools, and packages.

To address these problems, there has been considerable interest in improving the development, structure, and usability of scientific programs. The use of advanced software development techniques such as object-oriented programming is becoming increasingly common in scientific projects [31, 52, 86]. To provide better integration between tools, developers have been working on the creation of integrated problem solving environments and component frameworks [80, 84]. To improve usability, a number of efforts have focused on user interfaces and the way in which scientific programs are driven [80, 49]. If a sensible solution to these problems can be devised, it will greatly streamline the problem solving process as well as the way in which scientific programs are developed.

1.2 Technical and Cultural Challenges

Although there are many benefits to building better scientific software, solutions need to overcome a number of cultural and technical obstacles. Scientific computing is largely practiced by people trained in disciplines other than computer science. In addition, they generally pay little attention to software engineering and design. As a result, tools and techniques designed for large software engineering projects have largely been ignored by the scientific community. To be useful to scientists, solutions need to be easy to use and well adapted to the scientific computing culture. Furthermore, scientists are unlikely to abandon years of previous work or radically change their programming methodology in favor of unproven software technology. Therefore, tools must not only be simple to use, but they must work with a diverse range of software that is often idiosyncratic, difficult to use, and poorly designed.

1.3 The Need for Evolutionary Improvement

When faced with the prospect of improving scientific software, there is a tendency for software engineers to abandon existing scientific software and development techniques in favor of seemingly revolutionary improvements or new software technology. Unfortunately, this practice has the danger of producing a second-system effect in which a software environment is created with the goal of eliminating every possible shortcoming found in existing systems [17]. Unfortunately, users are often frustrated to find that such efforts result in systems that are too complicated and general purpose to effectively solve any problem.

Although improving the usability and structure of scientific programs is beneficial, it is important for software developers to realize that it is rarely necessary to throw existing software away and start over. In fact, many existing systems can be greatly improved by making a series of small modifications. Such an approach is attractive to scientists since they often develop a familiarity with their software and are reluctant to abandon previous work. Therefore, tools designed to improve scientific software are more likely to succeed if they embrace existing software and allow developers to make incremental and evolutionary improvements.

1.4 Scripting Languages

Scripting languages are a powerful tool for building better scientific software because they provide scientists with an interpreted environment that can be used to specify problems, control complex applications, and solve problems in an exploratory manner. In addition, scripting languages provide a framework for building and assembling software components. A component-based approach greatly improves the organization of scientific programs and allows different systems to be integrated. Such integration allows tools to work together more efficiently and streamlines the problem-solving process. Finally, scripting languages can interact with code written in compiled languages such as C, C++, and Fortran. This allows existing applications, as well as performance critical operations, to be incorporated as extensions to a scripting environment. This, in turn, provides an evolutionary path for improving the organization and use of existing software as described in the previous section.

The benefits of scripting languages have even led some researchers to make bold claims about the future. Paul Dubois writes,

> Much of scientific programming is exploratory in nature, and for that sort of programming the use of compiled languages will cease. Interpreters will simply be fast enough for most such calculations. More computationally intensive programs will be written as extensions of interpreted environments [33, p. 171].

Although scripting languages have much to offer computational scientists, it is unlikely that scientists will abandon the use of compiled code due to the computationally intensive nature of scientific applications and the relatively slow performance of interpreters (which is sometimes more than three orders of magnitude slower than compiled C or C++). Therefore, the integration of scripting environments with extensions written in compiled languages such as C, C++, and Fortran will be critical if scripting languages are to succeed in the computational science community.

Unfortunately, the incorporation of compiled code into a scripting environment is a difficult endeavor. This difficulty arises from the fact that scripting languages provide no automated mechanism for accessing compiled code. As a result, scientists are forced to write wrapper code that acts as a glue-layer between their application and the scripting language interpreter. Creating this wrapper code is complicated, tedious, and prone to error. Therefore, scripting languages currently require too much time and effort to be used with most scientific computing projects.

If the process of integrating scripting languages and compiled code can be simplified. computational scientists will be able to effectively utilize scripting in a wide range of applications. Such simplification has even been discussed in the literature. Paul Dubois also writes,

> The specification of information in order to run a significant physics calculation is a complex task; the use of scripting languages for making such specifications will become universal. We shall have good tools that automatically connect a scripting language to compiled modules [33, p. 171].

Although a number of existing tools can be used to create scripting language extensions, these tools are special purpose, limited in their capabilities, and somewhat difficult to use. As a result, these tools have remained of limited use to the computational science community.

1.5 Research Goals

The goal of the research is to develop a general purpose scripting language extension building tool and to demonstrate the impact of such a tool on the development, organization, and use of scientific software. In particular, the research will show how such a tool makes it easier for scientists to use scripting languages and how the use of a scripting environment fundamentally improves the way in which scientific software can be used to solve scientific problems.

1.5.1 Making Scripting Languages Simple to Use

The research will show how an automated extension building tool can simplify the way in which scientists currently utilize scripting languages. First, such a tool would allow scientists to easily retrofit existing applications with a scripting language interface. This would improve the usability of those applications and allow them to be used in a much more flexible manner than previously possible. Second, by automating the creation of scripting interfaces, scriptable applications would be largely insensitive to changes in the underlying implementation–making such applications more adaptable to change. Finally, by automating the process of extension building, scientists will be able to utilize scripting languages in situations where they might otherwise not be considered.

1.5.2 Simplifying Software Development

Scripting languages provide a highly flexible environment for controlling applications as well as integrating software components. If the construction of scripting interfaces can be sufficiently simplified, it will possible for scientists to easily incorporate software into a scripting environment. This, in turn, can have a dramatic impact on the continued development and organization of that software. In particular, the research will show how scripting languages lead to greater flexibility, better reliability, and improved modularity. Furthermore, it will be shown that such an approach allows different software systems to be packaged as collections of components and combined with other systems. This integration allows different programs to work together more efficiently than previously possible.

1.5.3 Increasing the Usability of Scientific Programs

Finally, the primary purpose of using scripting languages is to improve the usability of scientific programs. Scripting languages are particularly appropriate for scientific applications because they provide a flexible interpreted environment that can be used to specify complex problems, run simulations, and interact with programs in an exploratory manner. Currently, these qualities are usually only found in large commercial systems such as MATLAB, Mathematica, Maple, and IDL [53, 108, 22, 83]. However, the research will show that the use of extension building tools and scripting languages makes it easy for scientists to construct their own applications of comparable power and flexibility.

1.6 Methodology

A general purpose scripting language extension tool will be developed and freely distributed to the software development community. This purpose of this tool will be to automatically construct scripting language interfaces to existing programs written in C and C++. Although a large number of scientific programs are currently implemented in Fortran, the use of Fortran will not be considered. First, an increasing number of scientific programs are now being written in C or C++. Second, scripting languages require compiled extensions to be accessed through a C interface. At this time, the C interface to Fortran varies by compiler and is highly nonstandard (making automatic extension building difficult). Finally, since scripting language access to Fortran already requires a C interface, this interface can be used with tools designed for C and C++ code.

In addition to developing an extension building tool, a number of interface construction and design techniques for migrating existing applications to a scripting environment will be developed. Some of these techniques include methods for data management, error handling, type management, and the creation of scripting language components.

To demonstrate the impact of the tool and interface building techniques, a detailed case study will be conducted. The case study will describe the process of transforming an existing scientific program into a scriptable application and how that application improves as a result of operating in a scripting environment.

Finally, a user survey will be used to determine the effectiveness of the extension building tool with other applications. The survey will also help identify strengths and weaknesses of this approach as well as the impact on the application buildin process.

Results will be validated through the use of the case study and user survey. In particular, success will be based on the following criteria

Ease of use. Unless an extension building tool is easy to use, it is unlikely to be of much use to the scientific community.

Applicability to real software. To be successful, an extension building tool must be able to operate with the software developed and used by scientists.

Productivity. Tools must make scientists more productive by simplifying the development of scientific software and streamlining the way in which that software is used to solve scientific problems (i.e., improving the "usability" of scientific software).

Performance. Given that most scientific programs are computationally intensive, solutions must not introduce large performance penalties.

1.7 Results

A freely available scripting tool, SWIG (Simplified Wrapper and Interface Generator), has been developed and distributed [8, 5]. SWIG allows developers to create scripting interfaces to programs written in C, C++, and Objective-C. To simplify use, SWIG constructs scripting interfaces directly from ANSI C/C++ declarations as opposed to using a formal interface definition language. Thus, using only C header files, a scientist can often construct a simple scripting interface to an application in only a matter of minutes. In addition, SWIG has an extensible design that allows it to support multiple scripting languages and to be customized. Currently, SWIG is being used by several thousand users to construct extensions to Perl, Python, Tcl, and Guile on Unix, Windows, and Macintosh platforms.

In addition, a detailed case study is presented in which SWIG has been used to transform the SPaSM molecular dynamics code at Los Alamos National Laboratory into a highly flexible and efficient scriptable application [10]. In the process, the case study examines the use of SWIG and scripting languages with a real application over a 3-year period. As a result, the study provides a description of how an existing application can be incorporated into a scripting environment and how that application has improved over an extended period of time.

In the case study, it will be shown that SWIG enabled scientists to build a scripting interface to the SPaSM code in a relatively short amount of time and how the resulting scripting interface indirectly led to a series of incremental changes resulting in improved reliability, organization, and modularity. Furthermore, the use of SWIG and scripting languages eventually resulted in a high-performance highly flexible component-based system capable of integrated simulation, data analysis, and visualization. In addition, the scripting environment created with SWIG revolutionized the use of the code and made it possible for scientists to perform large-scale simulations of materials on a day-to-day basis.

Finally, a user survey consisting of 119 responses from current SWIG users is presented. The survey shows that SWIG is being used with a wide variety of scientific and nonscientific applications. Furthermore, survey responses indicate that SWIG greatly simplifies the creation of scripting language interfaces, improves productivity, and has a large impact on the development and use of C/C++ applications.

Based on the results of the case study and user survey, SWIG is shown to have positive impact on the development and use of scientific applications. First, SWIG greatly simplifies the integration of scripting languages and compiled code. This makes it possible to easily incorporate existing applications into a scripting environment as well as allowing scientists to use scripting languages in situations where they might otherwise have not been considered. Second, the use of SWIG and scripting languages simplifies the development and organization of scientific software–resulting in greater reliability, flexibility, and modularity. Finally, the use of scripting environments substantially improves the usability of scientific software and makes scientists more productive.

Even though this dissertation primarily focuses on the development of scientific software, SWIG is also applicable to other areas of software development. In particular, the user survey reveals that nearly 40% of SWIG users are working on nonscientific projects including industrial and commercial software development.

1.8 Organization

This dissertation primarily describes SWIG and the process of creating scriptable scientific applications. Chapter 2 describes some of the software problems faced by computational scientists and related research on scientific software environments. Chapter 3 describes scripting languages and the mechanisms by which they are extended with compiled code. The design and implementation of SWIG are described in Chapter 4. Chapters 5 and 6 describe strategies for migrating existing applications to a scripting environment as well as aspects of component-based scripting applications. Chapter 7 presents a detailed case study describing the use of SWIG and scripting languages with the SPaSM molecular dynamics code at Los Alamos National Laboratory. Finally, a user survey is presented in Chapter 8. This survey provides statistical data about who is using SWIG as well as antedotal evidence describing how SWIG simplifies the creation of scriptable applications, improves productivity, and improves the development and organization of scientific applications.

CHAPTER 2

SCIENTIFIC SOFTWARE

2.1 The Culture of Scientific Computing

Scientific computing has a unique culture that is quite different than that found in a commercial or industrial setting. In a nonscientific setting, the primary goal of a software project is usually the construction of a well-defined product such as a billing system, a CAD system, or a database. There are a variety of software engineering techniques that can be used to design, specify, and implement such projects. Furthermore, there are variety of metrics for measuring the success or failure of these efforts. The primary goal of most scientific projects, however, is not to build a specific product but to gain understanding and knowledge about a scientific problem of interest. Understanding this difference is important if successful tools are to be developed.

Most scientific computing projects are started by a small group of scientists (physicists, chemists, mathematicians, etc.) who are interested in studying a particular problem. More often than not, programs start small and are written to address a particular class of problems. Few computational scientists start with the goal of writing a large general purpose software package. However, programs that prove to be useful may evolve into larger systems over time.

When creating a scientific program, scientists are unlikely to use many (if any) of the software engineering methodologies that might be found in a large programming effort [109, 16, 17, 37]. The use of "requirements" documents, program analysis, CASE tools, and so forth is virtually unheard of. One reason for this is that scientific programs are almost always experimental and unproven. More often than not, the scientists may not know exactly how to solve the problem in advance. In fact, the entire "design" phase of a project may just be a discussion of the scientific problem (initial conditions, numerical methods, physical models, etc.). As a result, it is extremely difficult, if not impossible, to formally describe the structure that a scientific program will take in advance. Paul Dubois, writes:

> A scientific program is usually the product of one or two people, who write it initially to solve a class of problems faced by themselves and perhaps a few friends. It is much rarer for a decision to be made early to write a large program; rather, the programs that prove to be useful are added to, and evolve into, large programs over time. Such programs have not been suitable subjects for a massive analysis and design effort. In fact, scientists would not dream of doing such a thing even if they were to have the skill. Usually, it is not even known if the approach being taken will actually work. Anything remotely like a "Requirements" document is of questionable value to the scientist. Generally, the author has a class of problems in mind and an algorithmic idea that he or she believes will do the modeling job. The entire Requirements Phase usually involves a little muttering to oneself about what kinds of geometry and boundary conditions to allow for [29, p. 4].

Even though traditional software engineering techniques arguably might result in "better" scientific software, the inherently unpredictable nature of scientific problems makes the application of such techniques difficult. Genevieve Dazzo writes,

> Scientific programs tend to undergo more revision than their business counterparts because the needs of their users change more drastically over a short period of time. Users of scientific programs are anxious to explore new areas and expand existing knowledge [26, p. 52].

Finally, performance is an important part of the scientific computing culture and often one of the top priorities when developing scientific applications. Scientific problems routinely push the limits of available hardware and software. The need for performance is primarily motivated by the need for scientists to have an adequate turn-around time while still providing useful information about a problem. Simulations that take too long to complete are of limited value because they do not provide enough of a sample size to draw conclusions (simulations often need to be run dozens to hundreds of times with different parameters to be useful). Likewise, simulations that are of insufficient size may not have enough accuracy to yield interesting results. Interestingly enough, faster computing hardware does not seem to have had a large effect on simulation time. Rather, scientists have used increased computing power to improve the accuracy or size of their simulations. In fact, some authors have even observed that simulation times have remained relatively constant over the last 20 years despite huge gains in computing performance [29].

The performance focus of most projects does not necessarily mean that scientists ignore other software development issues. Portability is also a concern but is often not addressed until a machine is about to disappear. Making programs easier to use is also of interest, but not always a high priority. When these issues are considered, it is often within the context of performance. Solutions with severe performance penalties will usually be dismissed. However, scientists might also want to consider a quote attributed to John Ousterhout, "The best performance improvement is the transition from the nonworking state to the working state" [104, p. 447].

2.2 Scientific Software

The lack of formal design and piecemeal growth of scientific programs presents a number of technical challenges to framework and tool designers. Even though it is common for scientists to write software, they tend to do so by following the "principle of least action." In other words, scientists tend to favor techniques that are conceptually simple and require the least amount of effort on their part (although this phenomenon does not appear to be isolated to computational science). As a result, most scientific systems tend to be simple and minimalistic in nature.

Unfortunately, approaches that make a program easy to write can come back to haunt users and developers. For example, a program that starts small and is grown in an adhoc manner can become a nightmare to maintain. Likewise, a program that is easy to write might not be easy to use due to the difficulty of writing a user interface. This section describes some of the common problems associated with working with scientific software.

2.2.1 Piecemeal Growth

When a scientific program is first written, it usually addresses a specific scientific problem. For example, a program might be written to perform a three-dimensional molecular dynamics simulation of an elliptical crack in a periodic face-centered-cubic (fcc) crystal using a Lennard-Jones interatomic potential [3]. However, most programs can be generalized to look at other related cases so a scientist may start modifying the code with new boundary conditions, new interatomic potentials, a variety of numerical integration algorithms, and features for data management. When new features are added, conditional statements are often added to the program as follows:

```
if (boundary == FREE) {
    Use free boundary conditions
} else if (boundary == PERIODIC) {
    Use periodic boundary conditions
} else if (boundary == DAMPED) {
    Use damped boundary conditions
}
```

Even though adding new features to small programs is relatively simple, it becomes increasingly difficult as programs grow in size. In fact, after several years of this kind of development, scientists may find that a substantial portion of their program has become a tangled web of control logic, special cases, and obscure functions. Worse still, changing any part of the code may have far-reaching consequences and unforeseen side effects.

2.2.2 User Interfaces

Closely related to the growth and development of scientific software are the user interface mechanisms used to control such software. The most simple user interface is none at all. For small programs, parameters can be hard-coded into the program itself. This approach works fine for very simple problems, but scientific computing is an inherently exploratory activity. Scientists want to change parameters and see what happens. This becomes tedious if the code is recompiled after every change. An alternative approach is to modify the program to interactively prompt the user for various program parameters. This allows a user to change parameters at run time, but many scientific problems are solved by just changing one or two interesting parameters and observing the outcome of repeated simulations. Since answering the same series of questions quickly becomes repetitive, scientists eventually just write an input file containing the answers to all of the questions and run programs as batch processing jobs. Finally, scientific programs are sometimes controlled through a collection of command line options. However, users quickly become annoyed if they have to specify several dozen command line options each time a program is run.

Although all of these user interface schemes are easy to implement, they break down as programs grow in size and capabilities. As more features are added, the development of the user interface and the control of the program becomes increasing complex. At some point, it becomes unreasonable to explicitly ask the user hundreds of questions or to provide a hundred different options on the command line. The problem is futher compounded by the desire to integrate different packages and provide a more interactive problem solving environment. For example, none of the user interface techniques described so far would be appropriate for driving an integrated and interactive simulation, data analysis, and visualization environment.

The simplicity of existing user interfaces raises the question of why scientists don't use more sophisticated user interface strategies. One such strategy, often seen in scientific systems, is to utilize a simple command interpreter similar to what might be found in a commercial package such as MATLAB or Mathematica [53, 108]. Using an interpreter, a scientist would control an application by writing a simple script or typing commands that the application would interpret at run time. This provides a great deal of flexibility and appears remarkably similar to other techniques (especially since scientists are already accustomed to writing scripts and input files). However, making scientific programs interpret commands requires an interpreter. Writing a new interpreter from scratch is a time-consuming and difficult endeaver for scientists. On the other hand, using an existing interpreter can be equally difficult since a scientist may not know how to integrate it into their existing programs and use it effectively.

Finally, scientists might consider the use of a graphical user interface (GUI). This is often a "popular" notion until scientists discover the difficulty of creating a GUI. The development of a GUI is substantially more difficult than any of the schemes described so far–requiring detailed knowledge of graphical user interface libraries, event driven programming, widget libraries, and so forth. Furthermore, the implementation of a usable GUI is a nontrivial task. One would certainly not want to present the user with a dialogue box containing hundreds of buttons and entry fields because that would not be much different than just asking the user a series of questions. To further complicate matters GUI interfaces are often highly nonportable and difficult to manage on experimental platforms. In extreme cases, a machine might only support batch-processing jobs and have no support for graphical display. Finally, promoters of graphical user interfaces often assume that the user wants to constantly interact with their programs. Although interaction is clearly important, some scientific programs can run for tens to hundreds of hours. Therefore a scripting and batch processing capability is almost always necessary. As a result, a graphical user interface is most useful when combined with a command interpreter or other batch-oriented interface.

Overall, scientists tend to prefer user interface schemes that are simple to implement even though more sophisticated techniques are available. Since scientific programs start small, there is initially little need to utilize a highly sophisticated user interface. Furthermore, usability is only a minor concern since the initial developers of a system tend to be its primary users (also, the goal of most scientific projects is not to deliver a polished product). As a result, user interface problems tend to "sneak up" on developers as programs grow in size. In fact, it is not unusual for scientific programs to adopt a number of increasingly complex interface schemes over their lifetime.

2.3 The Search for Better Scientific Software

In later sections, the use of scripting languages and automated extension building tools will be described as a means for improving the usability and organization of scientific software. However, this is not the only approach being pursued in the scientific community. This section briefly describes a number of other development efforts. The primary goal of this section is to call attention to related work that is aimed at changing the way in which scientific software is developed and used.

2.3.1 Object-Oriented Frameworks

Some scientists have been adopting the techniques of object-oriented programming to provide an application development environment for solving science and engineering problems. Some efforts include POOMA, A++/P++, PETSc, and Diffpack [84, 81, 4, 18]. The idea behind these systems is to provide scientists with a useful collection of objects and an environment in which the objects can be used to solve problems. For example, a system might provide basic objects for matrices, unstructured meshes, vectors, complex numbers, particles, vector fields, and so on. A number of operations and methods such as basic arithmetic, linear solvers, preconditioners, visualization, and error analysis could then be applied to the objects as needed. To solve a problem with one of these systems, a scientist assembles an appropriate collection of objects and applies a series of "interesting" operations to them.

This approach is attractive for a number of reasons. First, it provides a tightly integrated environment that allows objects to interact with each other. Second, it allows software designers to hide much of the complexity from users. For example, on a parallel machine, the parallelism could be hidden away in abstract base classes and lower levels of the framework. At the highest level, users might not even be aware of such parallelism or the technical details involving its implementation. In addition, this approach can result in very compact and "simple" formulations of scientific problems. For example, a scientist might be able to solve a problem by simply creating a few objects and writing a few mathematical equations. Operator overloading and other advanced language features can often hide much of the underlying complexity while greatly reducing the amount of code that must be written by the user. Finally, such approaches attempt to capitalize on the general benefits of object oriented programming including management of large software systems, controlling complexity, code reuse, and encapsulation.

2.3.2 Computational Steering

Computational steering is an emerging field that attempts to provide integration between simulation, data analysis, and visualization [49, 79]. User interaction is a key feature because steering systems provide scientists with a highly flexible and interactive data exploration and simulation environment. That is, they allow scientists to interact with their data in real time, guide simulations, and play out different scenarios.

Steering systems are primarily focused on the way in which a scientist performs and interacts with simulations. Much of the work is focused on issues of data locality, moving data between machines, visualization techniques, and mechanisms for presenting the data to the user. Some recent steering efforts include the SCIRun system developed at Utah, program instrumentation tools at Georgia Tech, and integration of visualization systems such as AVS with simulation codes [80, 100, 99, 98, 97, 21, 59, 44].

The interesting aspect of steering systems, is that in providing integrated simulation and visualization to the user, they also address complex software construction issues. In order to make a steering system work, the different subsystems need to be combined and controlled in an effective manner. In many cases, the components are third-party packages and libraries. Therefore, developers need to worry about the interfaces between modules, frameworks for combining and using modules, and the difficulties of using existing software. Since these issues also arise in scripting environments, many of the techniques utilized by steering systems also apply to scriptable applications.

2.3.3 Heterogeneous Computing

A number of researchers have been interested in the problem of providing software and infrastructure for heterogeneous computing. Some efforts include the I-WAY, Globus, the Grid, and Legion [27, 40, 92, 47]. Significant portions of these projects are devoted to infrastructure issues such as faster networks, high performance computing platforms, and high-end visualization systems, but there is also a fundamental software problem that needs to be addressed. In particular, how are scientists going to go about hooking all of these pieces together? How will they write software to run in such a heterogeneous environment? How can existing systems be incorporated into such an environment?

Like efforts in computational steering and scripting environments, success depends upon finding schemes for building, controlling, and using scientific software components.

2.3.4 Computational Proxies

The integration and control of scientific software components have also been accomplished using object-oriented databases and computational proxies [24]. With a proxy system, the original scientific applications remain unmodified while a proxy system is used to provide a generalized interface to users. The proxy system manages the execution and transfer of data between different components while hiding details from the users. In order to do this, the proxy server knows how each program is controlled as well as the data formats used for input and output.

The proxy approach is primarily used to encapsulate a variety of legacy applications into a unified environment. It does not change the way in which each individual application is structured or used nor does it address the problems of moving massive amounts of data around between subsystems (although it may hide the process from users).

An approach similar to computational proxies can sometimes be accomplished using scripting languages. For example, Expect is a Tcl-based extension that is often used to drive existing applications by mimicking the input of users [63]. Likewise, Perl and Python can be used to drive legacy applications from a scripting environment [101, 66].

2.3.5 Components and Distributed Objects

The creation and integration of software components are also of great interest to commercial and industrial software development efforts. The primary difficulty in this case is that programming projects are often undertaken by large teams of programmers who are working on very large and complex systems. Since individual components may be developed by different groups of programmers, frameworks for integrating these components are of critical importance. Two of the most common component architectures include CORBA (the Common Object Request Broker Architecture) and Microsoft COM (Common Object Model) [74, 87].

CORBA is a specification created by the Object Management Group (OMG), a consortium of computer companies including Sun, HP, DEC, and IBM. COM is a competing component architecture developed by Microsoft and is the basis for most applications developed in the Windows environment.

When using COM or CORBA, applications are built by assembling components. Each component can be thought of as providing a specific service such as access to a database, performing computational intensive operations, or presenting the user with an interface. Although these services may all exist on a single machine, they may also be distributed across a network of machines. Thus, a database server could provide database access to other machines on the network and be used as a component in various other software packages.

Component architectures allow components to be completely decoupled, written in different languages, or to exist on different machines. However, the key to using CORBA and COM is that the interfaces between components are precisely defined. Interfaces are specified using an interface definition language (IDL) such as CORBA IDL. The IDL specification provides a language and platform independent description of all of the available objects, datatypes, and operations supported by a component. With an IDL compiler, the interface description is turned into client and server stubs that must be written by the developer. After a developer fills in these stubs, the component can be made available for general use by other software clients.

Even though CORBA and COM are being used in an increasing number of commercial applications, these systems are unlikely to have a large appeal to computational scientists since they are viewed as being too cumbersome and difficult to use in a scientific setting. The primary benefit that scientists would receive from a component architecture is a well-defined mechanism for gluing software components together. However, given the nature of scientific software and the culture of scientific computing projects, this task can often be accomplished through other means such as object-oriented frameworks or scripting languages.

2.4 Limitations of Other Approaches

Although many of the approaches described improve scientific software, they also suffer from a number of drawbacks that has prevented their widespread use in the scientific computing community. This section briefly describes some of these problems.

2.4.1 Poor Performance

Scientific applications routinely push the limits of the machines that they run on. Yet, object-oriented frameworks and component architectures have a number of well-known performance problems. In C++, if objects are created through inheritance, there is a performance penalty due to virtual function calls. Operator overloading and other advanced features often result in the creation and destruction of large numbers of temporary objects [28]. The creation of temporaries is also problematic for very large objects such as million element arrays (especially when memory utilization is critical). Component architectures such as CORBA suffer additional performance penalties since they are often built around RPC-like mechanisms for invoking procedures and methods.

A widely cited article in 1994 reported results in which C++ was as much as 700% slower than Fortran [50]. As a result, there has been considerable interest in techniques designed for improving C++ performance. One highly publicized technique involves the use of expression templates [51, 86, 95, 96]. Using expression templates, run time performance comparable with C and Fortran can be achieved for certain operations [95]. However this performance improvement is achieved by expanding arithmetic expressions into nested template definitions. This grossly inflates compilation time and makes debugging extremely difficult since most debuggers do not fully support templates. Given the rapidly changing and experimental nature of scientific applications, this is an unacceptable solution to many computational scientists.

In criticizing object-oriented of frameworks, it is important to point out that differences in design have a large impact on performance and that not all frameworks suffer from performance problems. The state of C++ compilers also appears to be improving [86].

2.4.2 Closed Systems

Most frameworks enforce a rigid set of rules that must be followed by software developers. For example, an object-oriented framework typically provides an extensive inheritance hierarchy that must be used by developers when developing new code. Likewise, component architectures such as CORBA and COM precisely define the mechanisms by which software components are constructed and interact with each other.

Although the formality provided by these approaches may be appropriate for large programming efforts, it also results in closed systems that complicate the use of existing software. For example, if a scientist wanted to incorporate an existing application into such an environment, they would be forced to encapsulate the application inside an adapter class that was compatible with the target framework [43]. Depending on the nature of the original application, this process could be quite complicated. Closed systems also discourage reuse since the components of one system are generally not usable within other systems. For example, COM and CORBA components cannot be easily used together. Likewise, C++ systems based on implementation inheritance make it almost impossible for components to be extracted and used in other systems.

2.4.3 Programming in the Large

Many frameworks are designed for large-scale programming efforts and the development of packages. For example, at Los Alamos National Laboratory, a recent article about the ASCI (Accelerated Strategic Computing Initiative) project stated "We will be forming code development teams larger than any we have ever attempted to manage, with as many as 20 to 30 staff members each. And these teams will be developing extremely complex software that must run on the world's largest massively parallel computers"[82, p. 3]. Rather than attempting to investigate new scientific problems, these efforts are primarily oriented towards developing production software for solving engineering problems. The formality provided by frameworks is likely to be a suitable mechanism for managing such projects. However, most computational scientists rarely set out to create massive software packages. As a result, the formalism and methodology used to manage large software projects often becomes an obstacle in small projects.

2.4.4 Poor Adaptation to Change

Frameworks tend to enforce a particular design model on users. Scientific programs, on the other hand, are usually grown in a piecemeal and adhoc manner. This difference creates two fundamental problems. First, a system that is too rigid may be too difficult to modify and extend with new features (or so difficult to understand that scientists do not know where to start). Likewise, when changes are made, they may be difficult to incorporate. For example, in a component architecture, the addition of new features would require modifications to interface definition files, regeneration of stubs, and so forth. In a scientific setting where software changes rapidly, this clearly presents a problem.

2.4.5 Conceptual Difficulties

Finally, for a scientist who has only written Fortran or C programs, jumping into a large object-oriented framework can be overwhelming. Not only must scientists learn a new language to use these systems, they need to learn a whole new vocabulary and mindset for thinking about problems. To further complicate matters, general purpose systems such as CORBA and COM are often bloated with features such as security, quality of service, garbage collection, version control, and fault tolerance. Few scientists have much interest (or need) to use such features and are easily overwhelmed by the complexity that they introduce.

2.5 Scripting Languages and SWIG

In the following chapters, the use of scripting languages and SWIG will be presented as a new approach for building, managing, and using scientific software. This approach is attractive because it solves many of the practical software problems encountered by computational scientists while addressing all of the above limitations. In particular,

Performance. Scripting languages can interact with code written in compiled languages. This allows performance critical operations to be easily written in C, C++, or Fortran.

Open systems. Rather than enforcing a rigid structure, scripting languages make it easy to work with a wide variety of software components. Furthermore, SWIG simplifies the process of incorporating existing packages into a scripting environment regardless of their underlying implementation or design.

Programming in the large and small. Although scripting languages have been used in large-scale programming projects, they enforce very few rules and can be easily used in small projects. This makes them appropriate for a wide variety of scientific programs.

Adaptation to change. Using extension building tools such as SWIG, scripting languages can easily respond to rapid changes in the underlying implementation of scientific programs. Furthermore, scripting languages can be easily used with programs that are under development or in an unfinished state.

Conceptual simplicity. Scripting languages are simple to learn and use. Furthermore, they can be added to the software already being used by scientists. Thus, scripting is more of an evolutionary improvement rather than an revolutionary change.

In addition, it will be shown that scripting languages and SWIG enable scientists to achieve many of the same benefits associated with other approaches. This includes improved modularity and encapsulation, systems integration, development of component architectures, and interactive exploratory problem solving.

CHAPTER 3

SCRIPTING LANGUAGES

Scripting languages have much to offer scientists because they provide a powerful mechanism for specifying scientific problems, integrating software components, controlling scientific systems. Furthermore, scientists already use simple scripts and scripting languages for a number of other tasks. This section discusses scripting languages, the benefits they bring to scientific computing applications, and the methods by which scripting languages are extended.

3.1 What Is a Scripting Language?

It is surprisingly difficult to give precise definition of a scripting language. However, scripting languages share a number of qualities.

Component gluing. Rather than building programs from scratch, scripting languages are primarily designed to glue components together. For example, the Unix shell provides an environment for executing and controlling programs as well as moving data between programs using files and pipes. In a similar spirit, scripting languages also can be used in a more fine-grained manner by gluing software libraries together, passing data between individual functions, creating collections of widgets for user interfaces, and so forth.

Interpreted. Unlike compiled languages such as C, C++, or Fortran, scripting language programs are interpreted. This eliminates the need for a separate compilation step and allows scripting languages to be run interactively.

High-level. Scripting languages provide a variety of useful data structures along with techniques such as dynamic typing. This results in programs that are smaller and easier to develop than in compiled languages.

Traditionally, scripting languages have been dismissed as being too simplistic to solve real problems. In fact, almost anything that can be done in a compiled language can also be accomplished in a scripting language. Many modern scripting languages also support object-oriented programming as well as aspects of functional programming found in languages such as Lisp and Scheme [94, 42]. In addition, most scripting languages also provide high-level access to operating system services such as the file system, sockets, and threads.

Much of the confusion regarding scripting languages is due to a misunderstanding of the role scripting languages play in relationship to systems programming languages such as C and C++. John Ousterhout writes,

> System programming languages were designed for building data structures and algorithms from scratch, starting from the most primitive computer elements such as words of memory. In contrast, scripting languages are designed for gluing: They assume the existence of a set of powerful components and are intended primarily for connecting components [77, p. 23].

3.2 Component Gluing

One of the most powerful features of scripting languages is their ability to glue software components together. John Ousterhout writes,

> I concluded that the only hope for us was a component approach. Rather than building a new application as a self-contained monolith with hundreds of thousands of lines of code, we needed to find a way to divide applications into many smaller reusable components. Ideally, each component would be small enough to be implemented by a small group, and interesting applications could be created by assembling components. In this environment it should be possible to create an exciting new application by developing one new component and then combining it with existing components.
>
> The component based approach requires a powerful and flexible "glue" for assembling the components, and it occurred to me that perhaps a shared scripting language could provide that glue [76, p. xviii].

The nature of scripting language "components" can vary widely. At a minimal level, a component might be a stand-alone program and a scripting language used for job control as found in a Unix shell. Packages such as Expect can also be used to script executables and mimic the input of users [63]. However, most scripting languages can also be extended with functions written in compiled languages such as C, C++, and Fortran. In this role, scripting languages can be used to interact with compiled libraries and programs at a functional level. This makes it possible to use scripting languages as a framework for interacting with compiled code and building software components.

3.3 High-Level Programming

An important aspect of using scripting languages is their support for high-level programming. To understand this, it is helpful to contrast scripting languages with low-level systems programming languages like C, C++, and Fortran. In compiled languages there are a few basic datatypes, a set of basic operations, and programming constructs such as loops, control flow, etc. Programs and data structures are generally built from scratch using these primitive features. Scripting languages, on the other hand, supply a rich variety of objects such as lists, associative arrays (i.e., hash tables), arrays, infinite precision integers, and so forth. They also assume the existence of a large set of components. Thus, rather than building applications from scratch, scripting languages allow applications to be built by gluing different components together and managing data with powerful data structures.

A second area where scripting languages differ is in their treatment of datatypes. Languages such as C and C++ have strict a type-checking mechanism that checks the validity of code during compilation. Violations of the type system result in compile-time errors. Scripting languages, on the other hand, defer type-checking until run time. Thus, the Python function

```
def add(a,b):
    return a+b
```

can be used for any two objects that can be legally added. For example,

```
>>> add(3,4)                # Integers
7
>>> add("Hello","World")    # Strings
HelloWorld
>>> add([3,4,5],[6])        # Lists
[3,4,5,6]
>>>
```

Dynamic typing also benefits component gluing because it makes it possible to combine and utilize components and objects in a way that is simply not be possible (or easily implemented) in a compiled language. To illustrate this, consider the following Python function:

```
def plot_data(x, y, npoints, color, img):
    for i in range(0,npoints):
        img.plot(x[i],y[i],color)
```

This function would work properly with any kind object that defined a "plot" method. This would be checked at run time and the use of an object without this method would simply result in a run-time error. In contrast, the strongly typed nature of C++ would greatly restrict the use of a similar function by forcing it to only operate on a specific types of objects or objects derived from a common base class. As a result, systems progamming languages tend to be much more rigid and formal with respect to the use of objects and the mechanisms used to glue components together. This often makes it more difficult to glue components together and reuse software components.

Critics are quick to point out that run-time checking can lead to hidden errors because errors are not detected until code is actually executed. Although this claim has some merit, run-time typing often results in code that is easier to write, more flexible, and highly reusable. Run-time checking has also been used successfully in other object-oriented languages such as Objective-C or Smalltalk [23, 46].

Finally, scripting languages excel at simplifying complicated programming tasks. For example, consider the process of writing a graphical user interface. If written in C or C++, it can take hundreds of lines of code to open a window and place a button on the screen. In contrast, this is easily accomplished with a simple two line Tcl/Tk script [77].

The high-level nature of scripting languages make it easier to develop significant applications in a short amount of time. In fact, recent reports confirm this fact by citing huge reductions in code size and development time [77]. The effectiveness of high-level languages has also been described by Frederick Brooks in the *Mythical Man-Month*:

> Surely the most powerful stroke for software productivity, reliability, and simplicity has been the progressive use of high-level languages for programming. Most observers credit that development with at least a factor of five in productivity, and with concomitant gains in reliability, simplicity, and comprehensibility [17, p. 186].

These benefits apply to the use of scripting languages in general but would clearly apply to scientific computing applications. In fact, the problems of traditional software development have already appeared in the scientific literature.

> Frankly, the limiting factor for future [scientific] systems may well be writing the software itself. Few hard, reliable data points exist for trends in software productivity, but the perception persists that productivity increases have been

glacially slow for programs written in conventional languages such as Fortran, C, Ada, or Java [85, p. 45].

Scripting languages may provide scientists with an alternative approach.

3.4 Scripting and Scientific Computing

Scripting techniques have already been used in a variety of scientific applications. Commercial systems such as MATLAB, Mathematica, Maple, and IDL provide interactive command-driven interfaces that are remarkably similar to scripting languages [53, 108, 22, 83]. A number of specialized languages such as Yorick and Basis have also been developed for building scientific applications [71, 32]. More recently, the Python scripting language has seen increased use in a variety of scientific applications [66, 30, 55, 13]. Scripting languages are also widely used in the tools used by scientists. For example, the Visualization Toolkit includes a Tcl/Tk interface [89]. Plotting packages, performance analysis tools, and computational steering systems such as SCIRun also make extensive use of scripting languages although this may not be apparent to the user [2, 48, 80]. In many cases, scientists may not be aware that their tools are using scripting languages in a substantial way.

To understand the benefits that scripting brings to these systems, consider the fact that many scientific applications are monolithic packages with limited flexibility. More often than not, they are controlled by a series of command line switches or a simple command processor. Furthermore, programs are typically used in a batch processing mode with little if any user involvement. Scripting changes this by encapsulating applications in a highly flexible interpreted environment. This provides a better mechanism for controlling scientific software and allows users to interact with programs and data. Not only that, scripting has a positive impact on the development of scientific software [32]. In particular,

Faster development. A surprising portion of many scientific applications is devoted to the handling of input parameters and control flow. Scripting languages already provide this kind of infrastructure. As a result, development can focus on the creation of modules, not the mechanism by which those modules are controlled. Systems in which scripting is applied may experience a reduction in code size [32].

Reduced debugging time. Scripting provides an interpreted and interactive environment for interacting with scientific programs. Scientists can query values, execute functions, and perform operations in a manner similar to that found in a debugger. If data analysis and visualization components are available, these can also be used in the search for bugs. Since this capability is always available, much less time is spent using debuggers.

Rapid prototyping. New features can often be implemented in the scripting language interface first and moved to compiled code later. Given the long compile times associated with many systems, having an interpreted development environment tends to reduce development time (since new features can be implemented and tested without recompilation).

Portability. Most scripting languages can operate on a variety of architectures including Unix, Windows, and Macintosh systems. By implementing an application within a scripting environment, cross platform support can be achieved with much less effort than before. This is because the scripting environment provides generalized support for platform-dependent operations such as I/O, graphical user interfaces, and process management.

Reuse. Scripting encourages the development of modular and reusable code. If a suitable collection of modules can be created, they can be reused in other applications.

Virtually every computational scientist has utilized packages that make use of interpreted interfaces. Furthermore such interfaces have proven to be highly successful in a variety of commercial systems. Therefore, it is surprising that scripting techniques are not used more frequently in scientific applications.

3.5 Scripting Language Extension Programming

Although scripting languages have a number of practical benefits, it is unlikely that scientists will abandon compiled languages any time in the foreseeable future. This is primarily because the performance of scripting languages is sometimes more than three orders of magnitude slower than a compiled language [88]. Despite the other benefits of scripting languages, they are not enough to offset the performance penalty that would be incurred by entirely giving up a compiled language like Fortran or C.

However, scripting languages can interact with compiled extensions written in C, C++, or Fortran. This largely eliminates the performance penalty by allowing performance critical code to be written in a compiled language and merely controlled through scripting. In such systems, the underlying application may rely upon high-performance numerical libraries while scripting languages would be used at the highest level of the system for control, problem setup, and user interaction. In this role, scripting languages only account for a tiny portion of the overall execution time while computationally intensive operations are still executed in compiled code and dominate the overall execution time. Therefore, the fact that a scripting language runs much times slower than compiled code may be of minimal concern.

3.5.1 Extension Modules

To extend a scripting language with compiled code, it is necessary to create an "extension module." An extension module consists of three parts as shown in Figure 3.1. First, there is the C/C++ code that implements the functionality of the module or which corresponds to an existing application that is to be incorporated into a scripting environment. Second, there is wrapper code that is used to provide the glue connecting the scripting interpreter and the underlying C code. Finally, there is a module initialization function. This function is used to register the contents of an extension module with the scripting language interpreter when the module is loaded.

When creating an extension module, it is necessary to write the wrapper code and module initialization function. To do this, scripting languages provide a C level API that developers can use to access the scripting interpreter, convert data to and from a C representation, report errors, register new commands, create variables, and so forth.

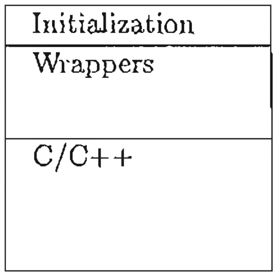


Figure 3.1. Extension module organization

3.5.1.1 Wrapper Functions

To execute functions and procedures in a compiled language, it is necessary to write wrapper functions. The role of a wrapper function is to convert datatypes between languages, provide the logic needed to make the function call, and to handle errors. To illustrate the process, consider a simple C function such as follows:

```
/* Compute n-factorial */
int fact(int n) {
    if (n <= 1) return 1;
    else return n*fact(n-1);
}
```

A wrapper function used to access this function from Tcl is shown below [76].

```
/* A Tcl Wrapper Function */
int
wrap_fact(ClientData clientData, Tcl_Interp *interp,
          int argc, char *argv[])
{
  int  result;
  int  arg0;
  if (argc != 2) {
    Tcl_SetResult(interp, "Wrong # args. fact { int } ",TCL_STATIC);
    return TCL_ERROR;
  }
  arg0 = (int) atol(argv[1]);
  result = fact(arg0);
  sprintf(interp->result,"%ld", (long) result);
  return TCL_OK;
}
```

For Tcl to access the wrapper function, it must first be registered with the Tcl interpreter. This is done in the module initialization function as follows:

```
/* A simple Tcl module initialization function */
int Example_Init(Tcl_Interp *interp) {
   if (interp == 0)
       return TCL_ERROR;

   /* Create a new command 'fact' */
   Tcl_CreateCommand(interp, "fact", wrap_fact, (ClientData) NULL,
                     (Tcl_CmdDeleteProc *) NULL);

   return TCL_OK;
}
```

When the extension module is loaded, the module initialization function is executed. This function registers a new command "fact" with the Tcl interpreter. When this command subsequently appears in a script, execution is passed to the wrapper function. The wrapper function collects arguments passed to the function and converts them to a C representation. Since Tcl passes all arguments as strings, the wrapper function converts arguments from strings to the appropriate C representation. After conversion, the real C function is executed. Finally, the return value of the the function is converted back into a string and returned to Tcl. Although the process has been illustrated for Tcl, a similar procedure is used for all scripting languages and detailed examples are shown in Appendix A.

3.5.1.2 Variable Linking

Variable linking is the process of accessing global variables in a compiled program from a scripting language. Even though the use of global variables is highly discouraged in software engineering circles, they are used quite frequently in scientific applications to store the values of various simulation parameters.

The simplest way to support global variables is through the use of functions such as the following:

```
// A global variable
double Dt;
...
// Get and set the value
double Dt_get() {
   return Dt;
}
void Dt_set(double d) {
   Dt = d;
}
```

These functions can then be added to the scripting interface as ordinary wrapper functions.

Some scripting languages, such as Tcl, provide an alternative mechanism that can be used to make global variables appear as ordinary scripting language variables. For example, executing the following C code in the module initialization function

```
Tcl_LinkVar(interp,"Dt", (char *) &Dt, TCL_LINK_DOUBLE);
```

turns Dt into a Tcl variable that is mapped directly onto a C global variable. When this variable is accessed or modified from the scripting interpreter, the underlying C variable is then accessed directly.

Other scripting languages can create special variables where read and write operations are mapped onto functions written in C. For example, in Perl, the following functions can be written.

```
int
wrap_set_Dt(SV* sv, MAGIC *mg) {
    Dt = (double ) SvNV(sv);
    return 1;
}
int
wrap_get_Dt(SV *sv, MAGIC *mg) {
    sv_setnv(sv, (double) Dt);
    return 1;
}
```

When a new value is assigned to `Dt`, the set method is used to change the value. When the value of `Dt` is read, the get method is used to retrieve the value. Thus, in a Perl script, `Dt` would appear, for all practical purposes, like an ordinary variable.[1]

```
# Change Dt
$Dt = 0.0001;             # Calls wrap_set_Dt

# Print out the value
print $Dt,"\n";           # Calls wrap_get_Dt
```

Support for variable linking varies widely between scripting languages. Global variables can always be accessed through a functional interface. However, if a scripting language offers an alternative mechanism, it can be used to make the scripting interface more convenient to the user.

3.5.1.3 Creating Constants

Most interesting programs, especially scientific ones, define a variety of constants for setting modes, physical constants, and so forth. In a C program, these might be defined as follows:

```
#define PI    3.14159265359
const double E = 2.71828182846;
```

[1] In Perl these are known as magic variables.

Making constants available to a scripting language interpreter can be accomplished by creating scripting variables that contain the corresponding value. This is done by placing special function calls in the module initialization function that create constants when an extension module is loaded. For example, in Python, placing the following function calls in the initialization function would create two constants

```
PyDict_SetItemString(d,"PI", PyFloat_FromDouble(PI));
PyDict_SetItemString(d,"E", PyFloat_FromDouble(E));
```

3.5.1.4 Object Manipulation

Although the interfaces to functions, variables, and constants are relatively straightforward, C structures, unions, and classes presents a more difficult problem. When working with objects, there are three fundamental problems. First, there is the issue of representation. Second, there is the problem of object creation and destruction. Finally, one must devise a mechanism for executing methods and operations on objects.

A common approach to the representation problem is to generate object handles. A handle is simply a name that is assigned to an object and used in the scripting language interface. Internally, a hash table is used to map handle names into pointers of the appropriate object type. When wrapper functions expect an object or pointer to an object, a handle name is used as a key in a hash table lookup. If a match is found, a pointer to an object is extracted and passed to the C function. If not, an error is generated.

To create and destroy objects, it is necessary to create and destroy handles. This is accomplished using special constructor and destructor functions that are added to the scripting language interface. For example, functions to create and destroy `Vector` objects might look like the following:

```
char *create_Vector() {
     Vector *v = new Vector();
     char *name = create_handle_name();
     add_handle(name,v);
     return name;
}

void delete_Vector(char *name) {
     Vector *v = (Vector) lookup_handle(name);
     if (!v) error("Not a valid object!");
     delete v;
```

```
        remove_handle(name);
        return;
    }
```

Although handles allow objects to be created, destroyed, and passed between different C/C++ functions, they do not allow a program to examine the internals of an object. Therefore, to invoke methods and extract internal information, accessor functions can be written. An accessor function provides a functional interface that can be used to manipulate objects given a handle. For example, if the definition of a `Vector` is

```
    struct Vector {
       double x,y,z;
       void normalize();
    };
```

the following accessor functions could be used to examine and modify member data.

```
    double Vector_x_get(Vector *v) {
         return v->x;
    }
    void Vector_x_set(Vector *v, double x) {
         v->x = x;
    }
```

Likewise, the following accessor function could be used to invoke a member function.

```
    void Vector_normalize(Vector *v) {
         v->normalize();
    }
```

Using accessor functions, access to objects is controlled entirely through function calls. As a result, a scripting interface can be built by simply creating wrappers around these function calls using earlier techniques.

Most modern scripting languages also provide support for object-oriented programming. An alternative approach to wrapping C and C++ objects is to encapsulate them with a scripting wrapper or adapter class. When a wrapper class is used, C and C++ objects are encapsulated inside a scripting language class. This class provides a natural object-oriented interface to the underlying objects and hides implementation details from users. For example, the following Python code illustrates the use of `Vector` objects when incorporated into a wrapper class.

```
    v1 = Vector()
    v1.x = 2
```

```
v1.y = 3
v1.z = 4
v2 = Vector()
v2.x = -1.5
v2.y = 4
v2.z = 5
v2.normalize()
d = dot_product(v1,v2)
```

The process of writing scripting language wrapper classes varies widely and is ommitted here for the sake of clarity. One approach, based on accessor functions, is discussed in Chapter 4. A variety of other object-oriented wrapping techniques can be found in [76, 66, 101, 106, 67, 39].

3.5.2 Compiling an Extension Module

To use a module it must be compiled in a form that the scripting language understands. Most modern scripting languages support dynamic linking of extensions [41]. With dynamic linking, extension modules are compiled into shared libraries or dynamic link libraries (DLLs). These libraries can then be loaded by the scripting language at run time. To load a module, a user simply starts the scripting language interpreter and issues a command such as "import foo." This command loads the module into memory as a shared library. Immediately after loading, the module initialization function is executed and control returned to the scripting interpreter. At this point, the contents of the module can be used.

Although supported on most machines, dynamic linking may not work in all cases. If building modules as shared libraries is not an option (or undesirable) it is also possible to integrate an extension module directly into the scripting language interpreter. To do this, the extension module and the scripting language interpreter are linked together to form a new executable. In the process, a new main program is written. This program initializes the scripting language interpreter and initializes the extension module upon startup. Thus, when the user runs the new version of the interpreter, the extension module will automatically be available for use.

3.6 Scripting Versus Commercial Packages

Many commercial packages such as MATLAB and IDL can be used as a framework for solving scientific problems [53, 83]. Not only do these systems have significant functionality, but they also have a foreign function interface. This allows a scientist to extend the package with new functionality and to utilize the functionality already provided by the system. For example, MATLAB can be extended with new functions by writing special wrapper functions in C [68]. In reality, the process of writing these wrappers is identical to that found with scripting language extensions.

In many respects, these packages can be viewed as domain-specific scripting languages. The system is controlled by an interpreted and interactive language that glues components together and can be extended by writing special wrappers (the same technique used by scripting languages). The main limitation of using commercial packages is their lack of generality and the fact that they are closed systems. For example, the only datatype supported in MATLAB is a matrix. This limited representation makes it difficult to represent nonmatrix objects and apply MATLAB to other domains.

Despite the limited generality of such systems, packages like MATLAB are examples of what a scriptable scientific application might look like–a collection of compiled modules controlled by an interactive and interactive language. Since such systems are so similar to scripting languages in both use and design, they will be included in further discussion. Thus, techniques described for extending Perl, Python, or Tcl could also be applied to a number of commercial scientific computing packages.

3.7 Scientific Computing and the Problems with Scripting

Despite the potential benefits that scripting languages offer scientists, they are not widely used in the scientific computing community. Although much of this may be due to a perception of poor performance, it is most likely due to the difficulty of integrating scripting languages with existing applications. It particular, there are the following problems.

The complexity of extension building. Building a scripting language extension is an extremely tedious and complex chore that requires an intimate knowledge of the target scripting language. Most scientists are simply not interested in this task.

The choice of scripting language. Given the complexity of building a scripting interface, the logical next step is to pick the "best" scripting language and use it for everything. Unfortunately, there is no such thing since all scripting languages have strengths and weaknesses depending on the application. For example, Tcl/Tk is primarily used in the construction of graphical user interfaces, Perl is used extensively for text processing, and Python for object-oriented programming. In many cases the choice of language may be a matter of personal preference. In any case, it is not inconceivable that one would want to use different scripting interfaces for different tasks. Unfortunately, the heavyweight extension mechanism all but prohibits this.

Rapid change. Scientific applications often change to address new problems. Unfortunately, the extension building process is not well-adapted to this environment since new features and changes to interfaces require changes to the underlying wrapper code.

Unless these problems can be addressed, it is unlikely that scripting languages will be of much use to scientists. Scientists must be convinced that scripting is simple to use and results in few performance penalties.

CHAPTER 4

SWIG

4.1 Compilation of Scripting Components

In this chapter, SWIG (Simplified Wrapper and Interface Generator) is described [5, 6, 8]. SWIG is a compiler that has been developed to automatically construct scripting language interfaces to compiled code written in C, C++, and Objective-C [61, 34, 23]. Versions of SWIG have been available for public use since February, 1996 and development has been ongoing. SWIG currently supports Perl, Python, Tcl, and Guile extension building on Unix, Windows-NT, and Macintosh systems [101, 66, 76, 65]. Experimental modules are also available for Java and MATLAB [38, 53].

This chapter is not intended to serve as a detailed description covering all of SWIG's features. Detailed information about using SWIG can be found in the *SWIG Users Manual* [9]. This chapter primarily focuses on the design, implementation, and operation of the SWIG compiler as well as a variety of associated language issues.

4.2 Related Work

Given the difficulty of building scripting extensions, there has been considerable interest in the creation of tools that simplify the task. Rather than writing glue code by hand, an extension building tool allows a user to specify the contents of scripting language component using an interface definition language (IDL). Interface descriptions are written in this language and compiled into scripting language components. Most scripting-related extension tools fall into the following categories :

Stub generators. A stub-generator compiles an IDL file into a file containing a collection of empty function definitions known as "stubs." The stubs contain all of the pieces needed to build a module, but it is up to the user to fill in the stub bodies with the appropriate glue code. Such a technique is most commonly found with distributed applications involving RPC, ILU, and CORBA, but can also found in scripting generators such as the Modulator tool used for building Python extensions [93, 25, 74, 66].

Language-specific module builders. Most scripting languages have specialized tools for building extensions. For example, h2xs and xsubpp are tools for building Perl extensions, Modulator can be used for building Python extensions, and Tcl has a number of tools such as Itcl++ and ObjectTcl [91, 66, 54, 106].

Application-specific generators. Large applications with scripting interfaces may include specialized interface construction tools. For example, the Visualization Toolkit (VTK) includes a YACC-based parser that compiles VTK C++ class definitions into Tcl, Python, and Java interfaces [67, 89].

Embedding tools. Embedding tools, such as Embedded Tk (ET) for Tcl, provide a mechanism for embedding scripting languages in compiled code [56]. This is a fundamentally different problem than controlling C/C++ code with a scripting language. Rather, these tools address the problem of accessing scripting languages from a compiled language.

Although extension building tools can simplify the interface generation process, they vary widely in capabilities and support. Most tools use their own interface definition format, making it nearly impossible to change tools or languages. In some cases, the use of a tool may even be nearly as difficult as writing an extension by hand. Finally, most extension building tools offer little in the way of documentation and support–often being labeled as obscure and magical tools for hackers and gurus. In fact, if one surveys popular scripting language books, almost no mention is made of such tools [76, 105, 101, 66]. This is unfortunate since the use of extension building tools greatly enhances the usefulness of most scripting languages.

Very little work appears to have been done in the development of general purpose scripting language extension tools that support both multiple scripting languages and a wide range of C/C++ code. The closest approximation is the interface builder packaged with the Visualization Toolkit, which is able to build to interfaces to Tcl, Java, and Python [89]. The ILU system also provides support for multiple languages, but is primarily used for distributed computing applications [25].

4.3 Design Goals

SWIG shares many of the features found in other interface generation tools, but attempts to address many of the limitations that make these tools difficult to use. Simply stated, the primary design goals of SWIG are as follows:

- Simplicity.
- Applicability to existing software.
- Support for rapid change.
- Separation of interface and implementation.
- Extensibility.
- Support for multiple scripting languages.

Meeting these goals involves a number of tradeoffs and considerations. For example, a tool that is simple to use might not provide the formality required in a very large software project. Likewise a tool that is too general purpose might not be able to produce quality interfaces to each scripting language. For a better understanding of the design, each goal is now described in some detail.

4.3.1 Simplicity

To computational scientists, a tool is simple to use if it requires a minimal effort to use effectively. In an ideal setting, tools designed to help scientists should not interfere with the problem solving process. In other words, the use of a software tool should not become the primary focus of a project. For scripting extension building tools, this can be achieved by fully automating the extension building process, making it as easy as possible for users to specify scripting interfaces, and to produce scripting interfaces that are closely mapped to the underlying compiled code.

To automate extension building, a compiler should produce a fully functional scripting language module, not a collection of stubs. Ideally, the user should not have to write any of the scripting wrapper code as described in Chapter 3 nor should they be required to modify the output of the compiler.

To simplify the specification of interfaces, a compiler should make it as easy as possible for users to seamlessly integrate scripting with their programs. One problem with many interface generation tools is their reliance upon special interface definition languages (IDLs) that require the user to precisely specify almost all aspects of their application. Although such an approach provides more formality and precision, it also makes such tools hard to use in the experimental and exploratory environment associated with scientific projects. In such cases, the development of the interface specification may be only slightly less cumbersome than writing wrapper functions by hand. Furthermore, the rapidly changing nature of scientific software complicates the maintenance of interface specifications and may result in situations in which interfaces are inconsistent with the actual implementation.

To simplify the specification of interfaces, the ANSI C/C++ declarations found in header files and source files could be used. By specifying interfaces in this manner, scientists would not have to learn a special interface definition language and would be able to quickly build scripting interfaces to existing programs. Such an approach also works well in a rapidly changing software environment since changes to the underlying C implementation are easily propagated to the scripting interface.

Finally, the scripting interfaces produced by the compiler should closely match the underlying C and C++ code. For example, a C function should be mapped to a scripting language command of the same name, variables mapped to scripting variables, and so forth. In other words, the scripting interface should merely be an extension of the compiled code. This is an important feature because computational scientists are most likely to work with both C/C++ code and scripts. Therefore, the scripting interface should merely expose the underlying functionality to the user in a straightforward manner as opposed to hiding or obscuring it.

4.3.2 Applicability to Existing Software

Scientific programs vary widely both in implementation and design. Furthermore, the implementation of such programs may be quite complex–utilizing sophisticated data structures and algorithms. To successfully build scripting interfaces, tools must support a wide range of programming styles and techniques. To accomplish this, the compiler must support a large subset of the programming features found in scientific programs including functions, global variables, constants, and classes. The compiler also needs to support a wide range of datatypes including fundamental types (integers, floating point, strings), structures and objects, arrays, and pointers. Finally, the compiler needs to be highly adaptable. Rather than requiring users to structure interfaces and components in a precise manner, it should be possible for users to add scripting interfaces to existing software without having to make substantial modifications to that software.

4.3.3 Support for Rapid Change

Scientific applications change more rapidly than their commercial counterparts. Interface generation tools must keep pace with this change without becoming a burden. The best way to support rapid change is to automate the interface generation process while making it nearly invisible to the user. By fully automating the compilation of scripting modules and using the same language syntax as the original application, interface generation can be hidden away in the compilation of a program. Thus, when changes are made to that program, they can automatically be reflected in the scripting interface.

4.3.4 Separation of Interface and Implementation

One problem with modifying existing applications to operate in a new environment is that those applications may be modified in a way that prevents their use in other settings. For example, if a scientist builds a Tcl interface to a scientific application by hand, there is a tendency for Tcl specific C code to creep into the original application. As a result, the program eventually becomes inseparable from its Tcl interface.

To prevent this, a compiler should strive to maintain a strict separation of the compiled code and its scripting interface. By doing so, the original application will remain general purpose and be usable in other settings (including those that do not involve scripting languages).

4.3.5 Extensibility

Just as scientific programs and problems change, the compiler should be extensible in order to handle new situations. There are two cases that need to be considered. First, a user may want to extend or alter the behavior of the compiler to provide a "better" interface to their program. Ideally, there should provide special directives or commands that can be placed directly in interface description files for this purpose. A second important area of extensibility is support for new scripting languages. A variety of scripting languages are currently available and new ones may appear in the future. Thus, the compiler should be general purpose and easily extended to support different languages as appropriate.

4.3.6 Support for Multiple Scripting Languages

When it comes to C extension building, scripting languages are surprisingly similar. They are all extended with wrapper code and the techniques for writing this wrapper code, building modules, and using extensions are essentially the same. A compiler that exploits this similarity and supports multiple languages has many interesting aspects. First, it largely eliminates the problem of choosing the "best" scripting language. Rather, different languages can easily be used and evaluated for the job at hand (or personal preference). Second, it allows applications to simultaneously support a variety of different interfaces. This generally improves the usefulness of an application and allows it to be used in a wide variety of different settings. Finally, a compiler supporting multiple scripting languages would unify a number of extension building efforts and provide a general purpose tool for building scriptable applications regardless of the scripting language being used. This, in turn, allows developers to focus their attention on the creation of scriptable applications, not the specific scripting language that will be used.

4.4 Implementation

SWIG is implemented in C++ and consists of three primary components: an ANSI C/C++ parser, a scripting language wrapper code generator, and a documentation generator as shown in Figure 4.1. The input to SWIG is a subset of the ANSI C/C++ language that is extended with special directives. The output of SWIG is a C or C++ source file that is compiled and linked with the rest of an application to create a scripting language module. The code generator and documentation generator are extensible to support different scripting languages and documentation formats respectively. Currently, scripting language modules are available for Perl, Python, Tcl, and Guile whereas documentation can be generated in HTML, plain text, and LaTeX. Further discussion will focus exclusively on the code generation process while details about the documentation system can be found in the *SWIG Users Manual* [9].

4.4.1 Parsing

The SWIG parser accepts a subset of ANSI C, C++, and Objective-C and is implemented using YACC [62]. Before parsing, all input files are passed through a C preprocessor that handles conditional compilation and macro expansion. In addition to normal C code, SWIG understands a number of special directives that are used to

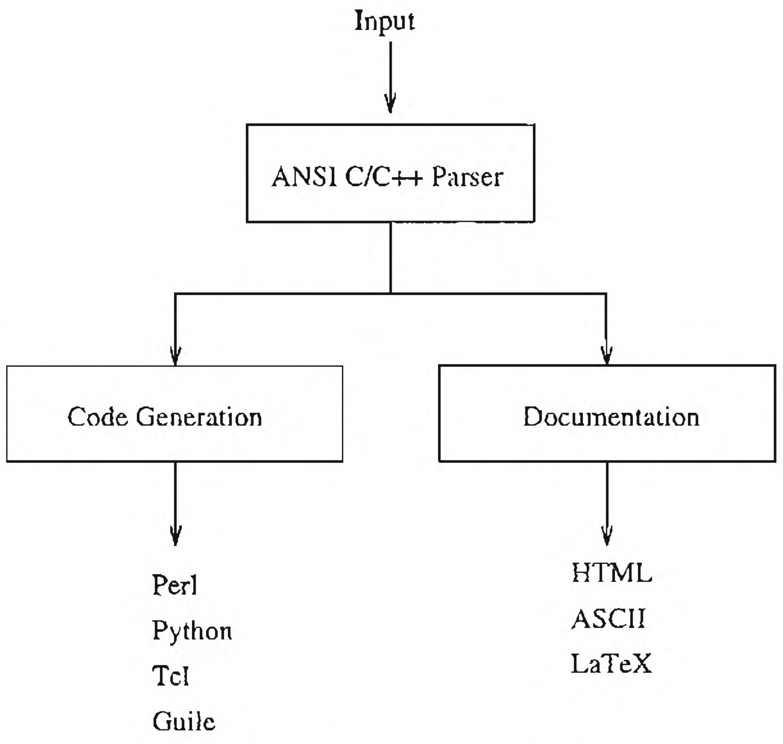


Figure 4.1. SWIG organization

guide the interface generation process as well as provide hints to the compiler. These directives are always preceded with a "%" to distinguish them from normal C keywords.

The parser only supports a subset of the C and C++ languages. Not all C constructs, such as pointers to functions, templates, and overloaded operators are easily mapped into a scripting environment. Thus, these features are currently ignored by the parser.

4.4.2 Code Generation

To support multiple scripting languages and to be extensible, SWIG defines a generic Language class with the following methods:

```
class Language {
public:
   ...
   virtual void set_module(name);
   virtual void create_function(name, returntype, parms);
   virtual void link_variable(name, type);
   virtual void declare_const(name, type, value);
   ...
};
```

To generate code, an instantiation of a particular language class is created (Tcl, Perl, Python, etc...) and given to the parser. The `set_module` method is used to set the name of the scripting language extension module. Afterwards, the parser executes methods such as `create_function`, `link_variable`, and `declare_const` to generate wrappers. To illustrate, suppose that the following C declarations were to be encapsulated in a module "Foo."

```
int fact(int);
void plot(Image *img, double x, double y, int color);
double Dt;
#define PI    3.14159265359
```

To construct the scripting language module, SWIG performs the following operations:

1. Create a new language object.

   ```
   lang = new LANG();
   ```

2. Set the module name.

   ```
   lang->set_module("Foo");
   ```

3. Create wrappers.

   ```
   lang->create_function("fact", int, (int));
   lang->create_function("plot", void, (Image *, double, double, int));
   lang->link_variable("Dt", double);
   lang->declare_const("PI", double, 3.14159265359);
   ```

4.5 SWIG Directives

Although the input to SWIG primarily consists of ANSI C/C++ declarations, a number of special directives are also available as shown in Table 4.1. These directives are used to guide the compilation process, provide hints, and customize SWIG's behavior. A full description of the directives can be found in the *SWIG Users Manual* although a brief description of the most commonly used directives can also be found in Appendix B [9]. A number of the more interesting directives will also be described in later sections.

Table 4.1. Commonly used SWIG directives

`%{ ... %}`	`%addmethods`
`%apply`	`%checkout`
`%clear`	`%disabledoc`
`%echo`	`%enabledoc`
`%except`	`%extern`
`%import`	`%include`
`%init %{ ... %}`	`%inline %{ ... %}`
`%module`	`%native`
`%name`	`%new`
`%pragma`	`%readonly`
`%readwrite`	`%rename`
`%typedef`	`%typemap`
`%wrapper %{ ... %}`	

4.6 SWIG Input Files

Since SWIG interfaces are built using a mix of ANSI C/C++ declarations and special directives, there are several approaches for constructing an input file. The most common approach is to use a separate "interface file." This file contains a selective list of the C/C++ declarations to be wrapped along with special directives. Another common approach is to insert SWIG directives directly into a C header file and to utilize conditional compilation. SWIG defines a symbol `SWIG` that can be used by the preprocessor for this purpose. Finally, SWIG can extract declarations directly from C source files.

4.7 A Simple SWIG Example

To use SWIG, the user specifies an interface using ANSI C declarations such as follows:

```
// file : example.i
%module example
%{
#include "example.h"
%}

int fact(int n);
double Dt;
#define PI  3.14159265359
```

To build the module, the user runs SWIG and compiles the wrapper code into a shared library as follows:[1]

[1]The compilation process varies according the compiler and operating system being used.

```
% swig -tcl example.i
Making wrappers for Tcl
% gcc -c -fpic example_wrap.c example.c
% gcc -shared example_wrap.o example.o -o example.so
```

To use the new module, the user starts the scripting language interpreter and loads the module as follows:

```
% tclsh
% load ./example.so
% fact 4
24
% set Dt 0.0001
% puts $PI
3.14159265359
%
```

To switch scripting languages, SWIG is given a different target language option. For example, a Python module could be built as follows:

```
% swig -python example.i
Making wrappers for Python
% gcc -c -fpic -I/usr/local/include/python1.5 \
  example_wrap.c example.c
% gcc -shared example_wrap.o example.o -o examplemodule.so
% python
Python 1.5 (#1, Jan  1 1998, 11:26:26)  [GCC 2.7.2.1] on linux2
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> import example
>>> example.fact(4)
24
>>> example.cvar.Dt = 0.0001
>>> print example.PI
3.14159265359
>>>
```

This simple example illustrates the use of SWIG and contains most of what users need to know to get started. First, interfaces are specified using ANSI C declarations. Second, a module name must be given using the `%module` directive. Header files and support code are then included using the `%{, %}` directive. Finally, the SWIG compiler is used to generate the wrapper code. This wrapper code is compiled and linked with the original application to create a module.

4.8 Datatypes and Data Representation

Although the processing of simple C declarations such as functions and variables is straightforward, the most difficult aspect of interface generation is the handling of C datatypes. In order to work with a wide variety of C code, the SWIG compiler must support most of the C built-in datatypes such as integers and floating point numbers as well as derived types such as pointers, arrays, structures, unions, and classes. Furthermore, methods for converting these types to and from a scripting representation must be devised. This section describes SWIG's treatment of datatypes.

4.8.1 Fundamental Types

ANSI C defines the fundamental datatypes shown in Table 4.2. The size of each type is implementation specific, but typical values for 32 bit architectures are shown. Scripting languages provide the datatypes shown in Table 4.3. When building the scripting language interface, SWIG maps the fundamental C datatypes into the closest appropriate scripting language datatype. This mapping is shown in Table 4.4. When datatypes are converted between C and scripting, truncation effects may occur. In particular, large integer values in a scripting language may be truncated when converted to a C datatype with less precision. Likewise, 64 bit C integers may be truncated when passed to a scripting language. In addition, single precision floating point numbers are usually cast to and from double precision values. Finally, two datatypes of notable interest are `char` and `char *`. In C, `char` is commonly used to hold a single character of text while `char *` is used to hold character strings. Therefore, both of these types are mapped into scripting language strings.

4.8.2 Pointers, Arrays, and Objects

Most C programs make extensive use of pointers, arrays, and data structures. To build useful scripting interfaces, a mechanism for handling these datatypes must be developed.

4.8.2.1 Typed Pointers

SWIG encodes C pointers as typed pointers in the scripting language interface. A typed pointer is simply a representation that contains both the value of the pointer and its corresponding type. For example, in Tcl, a C pointer of type "`Vector *`" might be encoded as the string "`_100fea8_Vector_p`." When typed pointer values are passed to C

Table 4.2. Fundamental C datatypes

Name	Description	Typical size (bits)
`int`	Integer	32
`long`	Long integer	32 or 64
`short`	Short integer	16
`float`	Single precision floating point	32
`double`	Double precision floating point	64
`char`	Single byte character	8
`void`	No value	-

Table 4.3. Scripting datatypes

Name	Equivalent C Datatype	Typical size (bits)
Integer	`long`	32 or 64
Float	`double`	64
String	`char *`	variable
None	`void`	-

Table 4.4. Datatype conversion

C Datatype	Scripting Datatype

`int`
`unsigned int`
`long`
`unsigned long`
`short`
`unsigned short`
`signed char`
`unsigned char`
	Integer (`long`)

`float` `double`	Floating point (`double`)

`char`
`char *`
	String (`char *`)

`void`	None

functions from scripting, the pointer value is extracted and the type compared against an expected value. If a type mismatch occurs, a run-time error is generated.

One of the primary differences between scripting and compiled languages is that scripting language defer type-checking until run-time. This differs from C and C++ where type checking occurs during compilation. With typed pointers, the type checking normally performed by a C compiler is performed at run-time using the type-signature information attached to each pointer value. When a violation of the C type system is detected, a run-time type error is generated.

Typed pointers provide a flexible mechanism for working with complex C datatypes. Essentially any C pointer value can be represented in the scripting language and used in a manner that is similar to C. The only major difference is that scripting languages are unable to dereference pointer values. In other words, scripting languages can represent and use C pointers but cannot peer inside or manipulate the objects that are being pointed to.

4.8.2.2 Arrays

Pointers and arrays are often used interchangeably in C programs since the "value" of an array in C is simply a pointer to the first element of the array [61]. Due to the close relationship between arrays and pointers, SWIG manages all arrays as pointers. By doing so, all of the techniques for manipulating typed pointers can be easily utilized.

Although the treatment of arrays as pointers is simple enough, there are a number of subtle problems with arrays. First, pointers contain no size information that can be used by scripting wrapper functions. Thus, a function such as

```
    void foo(double a[10]);
```

would accept any argument of type `double *` regardless of whether or not that argument was an array or an array of the proper size. Second, no special treatment is given to multidimensional arrays. When multidimensional arrays are wrapped by SWIG, a pointer to the first element of the array (stored in row-major order) is expected. Again, any argument of type `double *` may be used. Finally, there is no relationship between arrays in C and arrays in a scripting language. Although most scripting languages contain an array or list datatype, the representation of this data is different than that used in C. As a result, it is not possible to substitute a scripting language array for a C/C++ array (although SWIG can be customized to perform this conversion).

4.8.2.3 Structures and Objects

SWIG represents all structures and objects by reference using typed pointers. This avoids the problem of data representation because it is not necessary for SWIG or scripting languages to understand the internal implementation of a C/C++ object for it to be used. For example, the following SWIG interface could be used to provide access to a few functions in the standard C library,

```
%module stdio
%{
#include <stdio.h>
%}

// Some I/O functions
FILE     *fopen(char *filename, char *mode);
int       fclose(FILE *);
unsigned  fread(void *ptr, unsigned size, unsigned nobj, FILE *);
unsigned  fwrite(void *ptr, unsigned size, unsigned nobj, FILE *);

// Now a few memory allocation functions
void *malloc(unsigned nbytes);
void  free(void *);
```

From the scripting language interpreter, these function could be used in a completely natural manner. For example, the following Perl function copies a file using the above functions.

```
use stdio;
sub filecopy {
    my ($source,$target) = @_;
    my $f1 = stdio::fopen($source,"r");
    my $f2 = stdio::fopen($target,"w");
    my $buffer = stdio::malloc(8192);
    my $nbytes = stdio::fread($buffer,1,8192,$f1);
    while ($nbytes > 0) {
          stdio::fwrite($buffer,1,8192,$f2);
          $nbytes = stdio::fread($buffer,1,8192,$f1);
    }
    stdio::free($buffer);
    stdio::fclose($f1);
    stdio::fclose($f2);
}
```

In this example, the definition of FILE was not required to build the scripting interface, nor was it required to manipulate such objects from the scripting language interpreter.

Thus, SWIG allows scripting languages to manipulate a wide variety of C/C++ objects even when minimal information is available about the nature of those objects.

4.8.3 Unsupported Datatypes

SWIG supports most common C datatypes, but there are a few exceptions. The types of `long long` and `long double` are not supported because scripting languages do not provide enough precision to represent values of these types. Pointers to functions and pointers to arrays are not fully supported due to a limitation in the SWIG parser. Finally, pointers to C++ member functions are not supported since they have a different internal representation than other types of C and C++ pointers [34].

4.9 Objects, Classes, and Structures

As just described, SWIG represents all objects as typed pointers. Typed pointers contain no information about objects themselves so it is not possible to peer inside the object pointed to or to execute an object's methods. However, in Chapter 3, several methods for accessing objects were described. SWIG uses these techniques and provides a layered approach as shown in Figure 4.2.

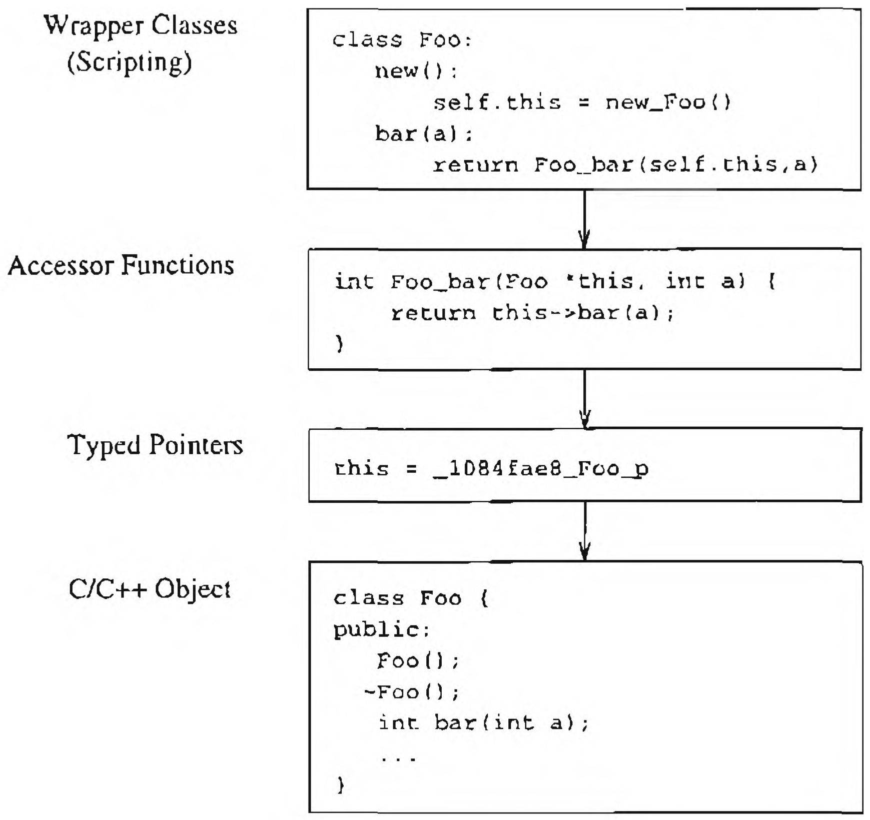
At the lowest level, typed pointers are used to represent objects. These pointers can be passed around between different C functions, but no further information is available. At the next level, accessor functions are used to look inside objects and execute methods. Finally, at the highest level, accessor functions are used to build wrapper classes that provide the user with a very natural object-oriented interface.

4.9.1 Objects as Typed Pointers

The representation of objects as typed pointers allows objects to be freely passed around between different C functions without regard for their internal representation. For example, the following functions could easily be turned into a scripting interface

```
#include "vector.h"
Vector *new_vector(double x, double y, double z);
void    delete_vector(Vector *v);
double  dot_product(Vector *v1, Vector *v2);
void    cross_product(Vector *v1, Vector *v2, Vector *result);
```

From the scripting language, these functions could then be used as shown in the following interactive session:



```
class Foo:
   new():
       self.this = new_Foo()
   bar(a):
       return Foo_bar(self.this,a)
```

```
int Foo_bar(Foo *this, int a) {
    return this->bar(a);
}
```

```
this = _1084fae8_Foo_p
```

```
class Foo {
public:
   Foo();
  ~Foo();
   int bar(int a);
   ...
}
```

Figure 4.2. Layered approach to objects

```
Python 1.5 (#1, Jan  1 1998, 11:26:26)  [GCC 2.7.2.1] on linux2
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> import vector
>>> a = new_vector(1,2,3)
>>> b = new_vector(4,5,6)
>>> print dot_product(a,b)
32
>>> result = new_vector(0,0,0)
>>> cross_product(a,b,result)
>>> print result
_100fe8a0_Vector_p
>>> delete_vector(a)
>>> delete_vector(b)
>>> delete_vector(result)
```

In this example, vectors are created and used in several functions. However, it is not possible to look inside a `Vector`. For example, when printing the result of the cross product operation above, only the typed-pointer value is returned.

4.9.2 Accessor Functions

To provide access to the internals of an object, SWIG automatically generates accessor functions when it is given the definition of a structure, class, or union. For example, the structure

```
struct Vector {
    Vector(double x, double y, double z);
   ~Vector();
    double x,y,z;
    void   normalize();
};
```

is expanded into the following collection of accessor functions:

```
Vector *new_Vector(double x, double y, double z) {
    return new Vector(x,y,z);
}
void delete_Vector(Vector *v) {
    delete v;
}
double Vector_x_get(Vector *v) {
    return v->x;
}
double Vector_x_set(Vector *v, double x) {
    return (v->x = x);
}
double Vector_y_get(Vector *v) {
        return v->y;
    }
    double Vector_y_set(Vector *v, double y) {
        return (v->y = y);
    }
    double Vector_z_get(Vector *v) {
        return v->z;
    }
    double Vector_z_set(Vector *v, double z) {
        return (v->z = z);
    }
    void Vector_normalize(Vector *v) {
        v->normalize();
    }
```

Since accessor functions are ordinary C functions, they can be wrapped into a scripting language interface using techniques described previously. When used in a scripting language, the user explicitly passes a typed-pointer to the accessor functions to extract information from the object or invoke methods.

Virtually any kind of C/C++ object can be manipulated through function calls in this manner. In fact, early C++ compilers used similar techniques to transform C++ classes into C code for compilation [34]. Because of the generality of this approach, the use of accessor functions forms a foundation for building scripting interfaces to most types of objects. Not only can accessor functions be used to interface with objects, they can be used from any scripting language (including those with no support for object-oriented programming). In scripting languages with object-oriented capabilities, the accessor functions can be used to build more sophisticated interfaces as described next.

4.9.3 Wrapper Classes

Using the accessor functions generated for objects, SWIG can optionally generate wrapper classes (also known as shadow classes). Wrapper classes provide a natural object-oriented interface around C/C++ objects using the object-oriented capabilities of the target scripting language. For example, in Python, a wrapper class might appear as follows:

```
    # A Python wrapper class
    class Vector:
        def __init__(self,x,y,z):
            self.this = new_Vector(x,y,z);
            self.thisown = 1
    def __del__(self):
        if self.thisown == 1:
             delete_Vector(self.this)
    def __getitem__(self,name):
        if name == 'x':
             return Vector_x_get(self.this)
        elif name == 'y':
             return Vector_y_get(self.this)
        elif name == 'z':
             return Vector_z_get(self.this)
        else:
     return self.__dict__[name]
    def __setitem__(self,name,value):
        if name == 'x':
             return Vector_x_set(self.this,value)
        elif name == 'y':
             return Vector_y_set(self.this,value)
        elif name == 'z':
             return Vector_z_set(self.this,value)
        else:
             self.__dict__[name] = value
    def normalize(self):
        Vector_normalize(self.this)
```

Using wrapper classes, objects can then be created and used as if they were objects created in the target scripting language. For example,

```
import vector
# Create some vectors
a = Vector(1,2,3)
b = Vector(4,5,6)
result = Vector(0,0,0)

# Compute some values
print dot_product(a,b)
cross_product(a,b,result)

# Print the result
print result.x, result.y, result.z

# Invoke a method
result.normalize()
```

4.9.4 Class Extension

When generating scripting interfaces to C/C++ objects, the interface does not need to exactly match that of the original object. In fact, object definitions can even be expanded with new methods and capabilities. For example, suppose that a user wanted to add a method for printing out the value of an object for debugging and diagnostics. This could be specified with SWIG as follows:

```
%addmethods Vector {
     void output() {
          printf("[ %g, %g, %g ]\n", self->x,self->y,self-z);
     }
}
```

When the scripting interface is built, SWIG will attach this new method to the original definition of Vector. As a result, it will be possible to use this method from scripting exactly as if it were part of the original object definition. For example,

```
>>> a = Vector(1,2,3)
>>> a.output()
[ 1.0, 2.0, 3.0 ]
>>>
```

The class extension mechanism only affects the scripting language interface and does not involve modifications to the original code or special C compiler tricks. Class extension turns out to be an extremely useful tool for building interfaces because C structures can be extended into classes, C++ classes can be extended with new methods, and programs can be made to appear object-oriented even if they are not.

It is important to note that class extension only affects the scripting language interface. Added methods are not visible to the original C or C++ program nor do they become part of the definition of an object. In fact, the primary purpose of class extension is to improve the scripting interface to objects. A further example of class extension will be given in Chapter 5.

4.9.5 Type Checking and Inheritance

When checking the type of a pointer, a comparison is made against an expected value. However, this presents a problem when working with inheritance hierarchies. To illustrate, suppose that a Shape class defined an abstract method for drawing as follows:

```
class Shape {
public:
     ...
     virtual  void draw() = 0;
     ...
};
```

When SWIG generates the wrappers for this class, the following accessor function is created.

```
void Shape_draw(Shape *s) {
     s->draw();
}
```

The accessor function expects a Shape object, but to operate correctly the function should allow any object derived from Shape to be used. To correctly capture this behavior, the SWIG run-time type checker is encoded with the C++ inheritance hierarchy. Thus, when extracting and checking pointer values, SWIG checks the type against the expected value as well as all derived types. In addition, proper type casting is performed to avoid slicing problems and to properly support multiple inheritance.

4.10 Type Management With Typemaps

The most critical aspect of extension building tools is the process by which different C datatypes are processed when generating wrapper code. In previous sections, general purpose rules for handling fundamental C datatypes and pointers were presented. However, SWIG also allows users to customize the way in which specific datatypes are processed. Such customization dramatically changes the nature of the generated scripting interface and allows users to tailor SWIG to the needs of their applications. This section provides a high-level introduction to typemaps and their use.

4.10.1 Typemaps

Simply stated, typemaps are special processing rules that are attached to specific C/C++ datatypes in order to customize the way in which SWIG generates wrapper code. The name "typemap" and general idea, has been derived from the xsubpp compiler packaged with Perl although the SWIG implementation expands upon the idea [91].

To illustrate how typemaps work at a high level, consider the following SWIG interface file:

```
%module example
%include constraints.i
%include typemaps.i

%apply double NONNEGATIVE { double px };
%apply int    *OUTPUT      { int *width, int *height };
```

```
/* Compute a square root */
double sqrt(double px);

/* Return the width and height of an image */
void   imagesize(Image *img, int *width, int *height);
```

Now, consider the use of the resulting scripting interface as shown for an interactive Python session.

```
>>> sqrt(-1)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: Expected a non-negative value.
>>> sz = imagesize(img)
>>> print sz
(400,300)
>>>
```

In this case, the sqrt function generates a Python exception when passed a negative value. Furthermore, the imagesize function takes the returned width and height and returns them as a two-element Python tuple.

To better understand what is happening, SWIG splits all C declarations into a collection of (type,name) pairs as follows:

```
(double,"sqrt")       -  sqrt: return type
(double,"px")         -  sqrt: Argument 1
(void,"imagesize")    -  imagesize: return type
(Image *,"img")       -  imagesize: Argument 1
(int *,"width")       -  imagesize: Argument 2
(int *,"height")      -  imagesize: Argument 3
```

The %apply directive in the interface file attaches special processing rules, known as typemaps, to specific (type, name) pairs. Thus, (double, "px") has been forced to be a nonnegative value whereas (int *, "width") and (int *, "height") have been marked as output values. During processing, SWIG checks each (type, name) pair to see if it matches any of the typemaps that have been specified. If a match is found, the special processing associated with the typemap is used when generating wrapper code. Once defined, typemaps apply to all future occurrences of a particular (type, name) pair. Thus, all occurences of int *width and int *height would be processed as output values in the above example.

4.10.2 Typemap Rules

So far, the general idea behind typemaps has been presented, but how are typemaps actually created? Consider the Tcl wrapper function for the factorial function given in Chapter 3.

```
/* A Tcl Wrapper Function */
static int
wrap_fact(ClientData clientData, Tcl_Interp *interp,
          int argc, char *argv[])
{
  int  result;
  int  arg0;
  if (argc != 2) {
    Tcl_SetResult(interp,"Wrong # args. fact { int } ",TCL_STATIC);
    return TCL_ERROR;
  }
  arg0 = (int) atol(argv[1]);
  result = fact(arg0);
  sprintf(interp->result,"%ld", (long) result);
  return TCL_OK;
}
```

The wrapper function performs several distinct operations. First, the function argument is converted from Tcl to C. Then, the C function is called. Finally, the result of the C function is converted back into Tcl. In SWIG, each of these operations is given a unique name such as "in" for input parameter processing, "out" for output value processing, and so forth. To define a new typemap, the `%typemap` directive is used as follows:

```
// Redefine the method for converting integers
%typemap(tcl,in) int n {
    $target = (int) atol($source);
    printf("Received n = %d\n", $target);
}
...
int fact(int n);
```

When SWIG generates wrapper code, the C code supplied in the typemap will be inserted into the wrapper function whenever a function argument of "`int n`" is encountered. In the process, the `$source` and `$target` tokens are replaced with the names of real C variables in the wrapper function. Thus, the wrapper function for `fact()` with the above typemap appears as follows:

```
/* A Tcl Wrapper Function with a typemap */
static int
```

```
wrap_fact(ClientData clientData, Tcl_Interp *interp,
          int argc, char *argv[])
{
  int  result;
  int  arg0;
  if (argc != 2) {
    Tcl_SetResult(interp, "Wrong # args. fact { int } ",TCL_STATIC);
    return TCL_ERROR;
  }
  /* Typemap code */
  {
     arg0 = (int) atol(argv[1]);
     printf("Received n = %d", arg0);
  }
  result = fact(arg0);
  sprintf(interp->result,"%ld", (long) result);
  return TCL_OK;
}
```

In this example, new C code (from the typemap) has been inserted into the wrapper function, replacing the original code that was used to convert integers from Tcl to C. The `$source` token has been replaced with `argv[1]` which contains the string Tcl passed as an argument. The `$target` token was replaced with `arg0` which is the integer value that will be passed to the real C function.

Although a simple example has been presented, SWIG defines approximately a dozen different typemap operations. These include input and output operations, default arguments, value checking, output arguments, and so forth as shown in Table 4.5. A full discussion of typemaps is not possible here and interested readers are advised to consult the *SWIG Users Manual* for more information [9].

4.10.3 Advantages of Typemaps

More formal interface generation tools often force users to explicitly state the nature of the interface being constructed. Therefore, functions might be declared as follows:

```
double foo(%input double *a, %output double *b, int n);
```

Unfortunately, annotating interfaces in such a manner is problematic. First, if users are required to annotate a large number of functions, it makes SWIG difficult to use. Second, such annotation breaks from ANSI C/C++ syntax. This would make it difficult to mix SWIG interfaces with C header files and would severely limit SWIG's ability to operate as rapid development tool. Finally, this approach would require the SWIG

Table 4.5. SWIG typemap rules

Name	Description
arginit	Initializes function arguments
argout	Returns values through function arguments
check	Checks the value of function arguments
const	Creates scripting language constants
default	Sets a default value to function arguments
except	Exception handling
freearg	Frees resources used in argument conversion
ignore	Forces an argument to be ignored
in	Converts values from scripting to C
memberin	Sets member data of C/C++ objects
memberout	Returns member data of C/C++ objects
newfree	Used to free memory
out	Converts data from C/C++ to scripting
ret	Cleans up return results of functions
varin	Sets global variables
varout	Gets the value of global variables

compiler to support a large collection of built-in processing rules. This would complicate the implementation of SWIG and limit its flexibility–especially if users did not like the behavior of the built in rules.

The typemap approach solves all of these problems. First, if programs uses consistent naming schemes for function parameters, typemaps can be used to quickly attach special processing rules to large collections of functions. In other words, once defined, a typemap applies to all future occurrences of a parameter avoiding the need to explicitly annotate every single function. Even if a program does not use a consistent naming scheme for parameters, it is unlikely that a developer would have picked names at random. Therefore, a SWIG interface file is often easily modified to fit into the typemap model. Second, the use of typemaps preserves ANSI C/C++ syntax. This allows users to customize interfaces while still being able to work with C header and source files. Finally, typemaps provide users with an almost unlimited number of customization options. Rather than rigidly defining a few special processing rules in the compiler, typemaps can be written to add almost any kind of special purpose processing. This makes SWIG more flexible and easily adapted to a wide variety of software.

4.11 Exception Handling

All scripting languages provide a mechanism for wrapper functions to report errors. Ideally, SWIG should exploit this to convert run time errors in the C/C++ code into scripting language errors. That is, if an error occurs someplace inside the C code, it should be reported to the user in the form of a scripting language error. SWIG allows users to specify exception handling code using the `%except` directive. For example, the following exception handler can be used to convert errors in the standard C library into Perl exceptions.

```
// A SWIG exception handler for the standard C library
%except(perl5) {
    errno = 0;
    $function
    if (errno) {
       croak(strerror(errno));
    }
}
```

During compilation, SWIG inserts the exception handling code directly into all of the wrapper functions. In the process, the `$function` token is replaced with the real C/C++ function call. As a result, a wrapper function might appear as follows:

```
XS(_wrap_fopen) {
    FILE *result;
    char *arg0;
    char *arg1;
    ...
    /* Exception handling code */
    {
       errno = 0;
       result = fopen(arg0,arg1);
       if (errno) {
          croak(strerror(errno));
       }
    }
    /* Return the result */
    ...
}
```

Should an error occur, an appropriate error message will now be extracted from the C library and reported back to the user as a Perl error.

4.12 Mixed-Language Programming Issues

The use of SWIG results in mixed-language applications in which scripting languages provide high-level control and compiled code is used to implement much of the underlying functionality. This section briefly describes some of the language issues that arise when working in such an environment.

4.12.1 Namespace Management

When incorporating existing applications into a scripting environment, it is sometimes possible to generate namespace clashes. First, the name of a C function or variable may conflict with the name of a keyword or function defined in the scripting language interpreter. To fix this problem, SWIG provides the `%name` directive that changes the name assigned to a declaration. For example, the declaration

```
%name(output) void print(char *s);
```

creates a new scripting language command "output" that is really mapped onto a C function "print." The second type of namespace clash occurs in the underlying C wrapper code created by SWIG. In rare instances, the C implementation of the scripting language may define symbols that are used by the application being wrapped. SWIG is unable to resolve these conflicts because they are due to linking problems (and outside the scope of SWIG's capabilities). However, these conflicts can usually be resolved by making minor modifications to the original application or with clever use of the C preprocessor. Fortunately, most scripting languages use a naming scheme that avoids these problems.

4.12.2 Memory Management

The use of scripting language extension modules involves the management of objects created by the scripting language interpreter as well as those created in C/C++. Given that scripting languages implement various forms of memory management and garbage collection, the use of C/C++ extensions raises a number of issues.

4.12.2.1 Garbage Collection and Pointers

SWIG manages all objects and complex data structures through the use of typed pointers. Although these pointers refer to some underlying C/C++ data structure, the data cannot be examined or directly manipulated by the scripting language interpreter. To further complicate matters, the scripting language interpreter has no way to know where the data being pointed to actually came from. Therefore, it would be a mistake for the scripting language interpreter to deallocate memory that was still in use in the underlying C/C++ application.

To prevent these problems, SWIG maintains a strict separation between the manipulation of typed pointers in the scripting interpreter and the underlying C/C++ data. Although scripting languages often implement garbage collection using reference counting, this is only applied to the pointer value itself—not the underlying data. In other words, when a pointer goes out of scope, only the pointer itself is deleted. To delete the underlying data, the user must explicitly destroy it by either invoking a C++ destructor or calling a deallocation function.

The explicit destruction of objects closely matches the memory management schemes used in C and C++ programs. For example, in C, objects are typically created and destroyed using `malloc` and `free`. In the scripting interface generated by SWIG, a program manipulating C/C++ data would be required to use an identical approach. Furthermore, just as in C, a program written in a scripting language would be subject to the same potential problems found in C programs such as memory leaks, accidental use of deallocated memory, dangling pointers, and so forth. That is, the use of C/C++ data from scripting is not much different than the use of that data from C or C++.

4.12.2.2 Implicit Memory Allocation

Certain C functions implicitly perform a memory allocation when executed. SWIG has no way to know if this occurs or not. Therefore, it is often up the user to know which functions allocate memory and to clean up that memory when it is no longer in use. In addition, SWIG provides a special directive, `%new` that can be used to provide a hint to the code generator that a function is returning newly allocated memory. For example,

```
%new char *get_message();
```

tells SWIG that this function is returning newly allocated character string. If the user chooses, they can define a typemap that cleans up this memory upon exit from a wrapper function. For example,

```
%typemap(newfree) char * {
   free($source);
}
```

In this case, the C function returns a newly allocated string that is copied into a scripting language string. Afterwards, the code supplied in the typemap is used to deallocate the returned memory before passing control back to the interpreter.

SWIG sometimes generates implicit memory allocation when returning objects by value. For example, the function

```
Vector cross_product(Vector *v1, Vector *v2);
```

returns a new object by value. Since SWIG only knows how to manipulate pointers, this function gets translated into the following wrapper code:

```
Vector *wrap_cross_product(Vector *v1, Vector *v2) {
    Vector *result = (Vector *) malloc(sizeof(Vector));
    *result = cross_product(v1,v2);
    return result;
}
```

In this case, every use of the function would result in an implicit memory allocation. It is up to the user to explicitly deallocate the result of the function by invoking `free`.[2]

4.12.2.3 Objects and Wrapper Classes

In Section 4.9.3, scripting wrapper classes were described. Wrapper classes provide high-level management of C and C++ objects. As a result, it is possible to support a limited form of garbage collection and data management. When SWIG creates scripting wrapper classes, it adds a special ownership attribute to the wrapper class. This attribute determines if the scripting language interpreter or C/C++ owns a particular object. When the interpreter cleans up a wrapper class object, it invokes the class destructor. This destructor examines the ownership attribute to see if the interpreter owns an object. If so, the underlying C/C++ destructor is invoked. If not, the wrapper class is destroyed, but the underlying object is preserved.

To determine ownership, SWIG applies a simple rule: if an object is created from the scripting language interpreter, it is owned by the interpreter. As a result, wrapper objects that are created from scripting are automatically managed by the interpreter while all other objects are managed by C/C++. In addition, the ownership of objects can be explicitly changed by the user if needed.

[2]In C++, the default copy constructor is used to copy the returned object.

4.12.3 Callbacks

Some C/C++ applications use callback functions to implement certain functionality. When working in a scripting environment, it may be desirable to implement callback functions in the scripting language interpreter itself. SWIG does not provide any built-in support for this type of programming. However, such a capability can often be implemented through the use of special wrapper functions and typemaps [9].

A closely related problem is that of overriding C++ virtual functions with methods written in scripting languages. Again, SWIG provides no builtin support for supporting this style of programming. However, handwritten wrappers can often be written to accomplish this.

4.12.4 Process and Resource Management

Scripting languages provide extensive support for managing operating system resources and processes. However, scripting languages do not have the ability to know what resources are utilized by an underlying C/C++ extension nor are extensions able to peer inside the interpreter. As a result, there is a separation between the resources used by the interpreter and its compiled extensions. Although this is rarely a serious situation, it can lead to problems in complex applications. For example, an extension that is multithreaded may cause the scripting interpreter to crash if it is not thread safe.

4.13 The SWIG Library

To encourage reuse and to make interface generation easier, SWIG is packaged with a standard library. The library consists of modules that provide scripting interfaces to common libraries (memory management for instance) as well as common customization options. Library files are included in an interface using the `%include` directive as follows:

```
%module example
%include pointer.i
%include exception.i
%include typemaps.i
...
```

The most powerful feature of the library is that it is designed to be independent of the target scripting language. Thus, many library files are written to work correctly with any of the target scripting languages. Finally, the library mechanism is a useful mechanism for working with large packages since interface files can be created and put in a shared repository. All of the users working on a system can then build interfaces to different components using files in the repository as needed.

4.14 Limitations

Even though SWIG attempts to simplify the construction of scripting interfaces to existing applications, it has a number of limitations. First, not all C/C++ datatypes are supported as described in Section 4.8.3. Second, the following C++ features are currently unsupported:

- Operator overloading.
- Overloaded functions.
- Namespaces.
- Templates.
- Nested classes.

Finally, some applications may be poorly suited for use in a scripting environment. For example, C++ programs making extensive use of advanced features may be difficult to incorporate in a scripting environment. Likewise, packages involving complex APIs may make scripted use and wrapper generation difficult.

Although limitations exist, most "typical" applications can be incorporated into a scripting environment with a little work. Even where limitations exist, a number of workarounds are available. Some of these workarounds are described in Chapter 5.

4.15 Summary

Although it is impossible to cover all of SWIG's features and implementation details, there are a number of important points. First, SWIG is designed to build scripting interfaces to existing applications written in ANSI C/C++. To simplify this process, interfaces are constructed using the header and source files of those applications. Second, SWIG provides mechanisms for interfacing with most of the datatypes and constructs that would be found in a typical C/C++ program. In addition, SWIG is highly adaptable and allows users to customize interfaces with exception handlers and typemaps. Finally, SWIG provides a library mechanism that can be used to simplify the construction of interfaces and encourage reuse.

CHAPTER 5

INTERFACE CONSTRUCTION

The effective use of SWIG generally requires more than simply grabbing a C header file and turning it into a scripting interface. Therefore, the construction of useful scripting interfaces may require the introduction of helper functions, changes to the way in which SWIG generates wrapper code, and minor modifications to the application itself. Furthermore, a user may decide to change the entire appearance of an application in order to promote usability and flexibility. This chapter describes the process and techniques used to build scripting interfaces to existing applications.

5.1 First Use of SWIG

A typical C/C++ application consists of functions, variables, and objects that form its implementation. In addition, there is a main function and control code that is used to start and drive the program. When first building a scripting interface, applications are transformed as shown in Figure 5.1. In this transformation, the underlying implementation remains largely unaffected while the control code is replaced with a scripting language interpreter. In general, the transformation process involves the following steps:

- Locate header files containing C/C++ declarations.
- Copy the headers to a separate interface file.
- Edit the interface file (if necessary).
- Remove the application's main() function and control code.
- Run SWIG to create a scripting language module.

Since commonly used datatypes and functions usually appear in header files, these files provide a good starting point for building the scripting interface. Although SWIG can sometimes work directly with header files, the contents of these files are usually copied

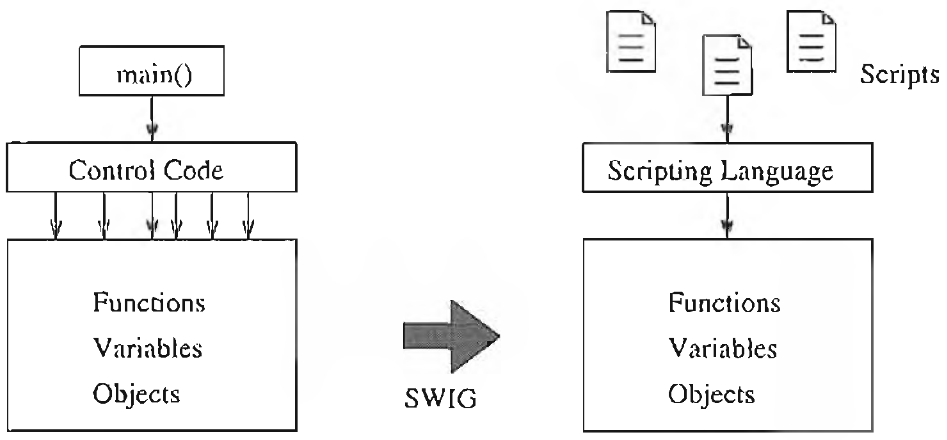


Figure 5.1. Creation of a scriptable application

to a separate interface file. This allows the user to remove problematic declarations and add SWIG directives as needed. Thus, the process of building a quick and dirty scripting interface really involves little more than copying a few files and making a few small adjustments.

After SWIG has been used to create a scripting interface, much of the underlying functionality of an application is exposed to the scripting language interpreter. When using the scripted version, users can interactively execute functions, set and query variables, and perform almost all of the tasks that might have been previously written in C. In fact, the application almost behaves as if it is running inside a debugger or some other development tool.

In many cases, the first use of SWIG can be a rapid process. In fact, one early user, who was new to SWIG, managed to create a Tcl interface to the OpenGL graphics library in approximately 10 minutes [75]. Such results are encouraging, but a word of caution is order. Even though it can be easy to create a scripting interface, that interface may be awkward to use or partially broken. Therefore, effective interface building generally requires a little more work.

5.2 Evolutionary Interface Development

SWIG promotes the evolutionary development of scriptable applications. Rather than requiring a scientist to precisely specify a complete interface in advance, header files can be used to quickly put a scripting interface on an existing package. By using the scripted version, limitations and problems can be identified. These problems, in turn, can be fixed by either modifying the SWIG interface description or the underlying application in an appropriate manner. This process then repeats itself as additional problems are discovered.

This approach is particularly well suited for scientific applications because it allows the underlying application to be used even if there are minor problems in the scripting interface. Thus, the application and its interface can be improved as the application is being used for practical work. In addition, the evolutionary approach closely matches the piecemeal manner in which scientific applications are typically developed and maintained. Finally, the evolutionary approach ultimately results in better interfaces because changes are motivated almost entirely by the use of the application. In other words, by using the application, techniques for improving the interface can be easily be found and implemented. This, in turn, results in an interface that is well suited to the needs of users and the problems at hand.

5.3 Helper Functions

To supply missing functionality or to provide a scripting interface with new features, "helper functions" can be added to a SWIG interface. A helper function is simply a new function, written in C, that is added to the scripting interface (and "helps" to improve the interface in a manner of speaking). A simple helper function is as follows:

```
%inline %{
void
print_Vector(Vector *v) {
    printf("[ %g, %g, %g ]\n", v->x, v->y, v->z);
}
%}
```

The `%inline` directive is used by SWIG to add new C functions to the scripting interface (the term "inline" is used because the given C code is inlined into the output wrapper code). In this case, a simple debugging function has been added to the interface to output the value of `Vector` objects.

Limitations in the SWIG parser can also be addressed using helper functions. For example, SWIG and most scripting languages do not support C++ operator overloading. However, an overloaded operator can be encapsulated in a helper function as follows:

```
%inline %{
Vector
vector_add(Vector &a, Vector &b) {
        return a+b;
    }
    %}
```

Helper functions can also be used to change the interface to certain parts of a package. For example, a scientific application might require the user to set the values of global variables before calling a function as follows:

```
    Min_x = 0.0;
    Min_y = 0.0;
    Min_z = 0.0;
    Max_x = 283.1;
    Max_y = 283.1;
    Max_z = 500.0;
    ...
    initialize_geometry();  # Implicitly depends on above parameters
```

Such an approach is somewhat awkward to use from scripting because there are implicit dependencies between the variables and the behavior of the function call. Therefore, a helper function could be used to provide an alternative interface as follows:

```
    %inline %{
    void geometry(double xmin, double ymin, double zmin,
                  double xmax, double ymax, double zmax) {
        Min_x = xmin;
        Min_y = ymin;
        Min_z = zmin;
        Max_x = xmax;
        Max_y = ymax;
        Max_z = zmax;
        initialize_geometry();
    }
    %}
```

Helper functions are an integral part of using SWIG because they supplement and enhance the scripting interface without affecting the underlying application. Even though the creation of helpers may appear tedious, they can almost always be written in ordinary C or C++ without regard for scripting language internals. As a result, helpers are relatively easy to write and are usable from all scripting languages.

5.4 Type Management

In Chapter 4 methods for representing various datatypes were described along with the SWIG pointer model. This section describes the manipulation of C datatypes within a scripting environment along with some of the techniques used to make better interfaces.

5.4.1 Type Conversion

By default, SWIG manages all types other than than the simple built-in C datatypes as pointers. Although simple, this approach can lead to usability problems by making the scripting interface awkward to use. As a result, it may be useful to override SWIG's default behavior and to process certain datatypes differently. For example, the OpenGL library contains a large number of functions that expect small arrays passed as arguments (params) such as

```
void glLightfv(GLenum light, Glenum pname, GLfloat *params);
```

By default, SWIG will generate an interface that requires the params argument to be passed as a pointer. To generate this pointer, helper functions can be written to create an array, populate it with values, and deallocate it when finished. However, a more elegant approach is to use a typemap to map scripting language lists and arrays into an appropriate params object. An example typemap for Python is as follows:

```
%typemap(python,in) GLfloat *params(GLfloat temp[10]) {
    int i, sz;
    if (!PyList_Check($source) {
       PyErr_SetString(PyExc_TypeError,"Expected a list!");
       return NULL;
    }
    sz = PyList_Size($source);
    if (sz > 10) sz = 10;
    for (i = 0; i < sz; i++) {
       PyObject *o = PyList_GetItem($source,i);
       if (PyFloat_Check(o)) {
          temp[i] = (GLfloat *) PyFloat_AsDouble(o);
       } else {
          PyErr_SetString(PyExc_TypeError,"Expected a float!");
          return NULL;
       }
    }
    $target = temp;
}
```

The precise implementation of the typemap is not so important for this discussion. However, the use of the typemap changes the scripting interface so that Python lists can be used wherever a parameter of GLfloat *params occurs in an interface. For example

```
glLightfv(GL_LIGHT0, GL_AMBIENT, [0.0, 0.0, 0.0, 1.0])
glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, [-1.0, 0.0, 0.0])
...
```

5.4.2 Containers

Sometimes it is useful to keep track of additional information about various C objects in a scripted application. For example, it might be useful to manage C arrays as both a pointer value and size to improve reliability. To illustrate, consider the following C function:

```
double
dot_product(double *a, double *b, int len) {
    int i;
    double result = 0.0;
    for (i = 0; i < len; i++,a++,b++) {
        result += (*a)*(*b);
    }
    return result;
}
```

In this function, two arrays are passed as pointers, but no guarantee is made about the size of those arrays. In fact, a size mismatch could result in a program crash or erroneous result. Unfortunately, there is no way for the function to know the real sizes of the array arguments.

One trick for working around this problem is to use a container structure. For example, a C array might be described by the following structure:

```
struct DoubleArray {
    double *ptr;
    int     size;
}
```

Using the container, a simple helper function can be written to override the original C function as follows:

```
// Create a helper for dot_product
%{
double new_dot_product(DoubleArray *a, DoubleArray *b) {
    if (a->size != b->size) {
         error("Array size mismatch!");
         return 0.0;
    }
    return dot_product(a->ptr,b->ptr,a->size);
}
%}

// Wrap the helper function (but use the original name)
%name(dot_product)
double new_dot_product(DoubleArray *a, DoubleArray *b);
```

In the scripting interface, the dot_product command will now expect two array objects as arguments. If any other kind of object is passed, or if the sizes mismatch, an error will be generated. Thus, in effect, the scripting interface is shielding the underlying C application from a potential error.

5.4.3 Aliasing

SWIG provides minimal support for certain C/C++ datatypes such as pointers to functions and templates. Some of these difficulties are due to parsing limitations (very complex datatypes) and others are semantic (what is a template in a scripting language?). However, some of these problems can be eliminated by hiding problematic datatypes behind new names. To illustrate, consider the definition of a simple C++ template class.

```
template<class T> class List {
private:
...
public:
    List();
   ~List();
    void append(T obj);
    T    get(int n);
    T    remove(int n);
    int  length();
    ...
};
```

If this class were given to SWIG, it would be ignored since there is no way to build a scripting interface to a raw template definition (since no real type-information is available). However, an aliasing trick can be used to produce an interface to a specific instantiation of the template class as follows:

```
// SWIG interface to a C++ template instantiation
%{
// Insert a typedef into the wrapper code (SWIG ignores this)
typedef List<double> DoubleList;
%}

// Wrap it as a normal class
```

```
class DoubleList {
public:
    DoubleList();
   ~DoubleList();
    void   append(double obj);
    double get(int n);
    double remove(int n);
    int    length();
    ...
};
```

In this case, the template instantiation has been aliased to a new name of `DoubleList`. This name is then used to define a class in the interface file. When compiling, SWIG converts the class definition into wrapper functions and produces a scripting interface capable of creating and manipulating objects of type `List<double>`.

5.5 Object-Based Interfaces

Although the use of objects is supported (to varying degrees) by most scripting languages, existing applications written in C may make limited use of such techniques. However, SWIG can be used to retrofit an object-based scripting interface onto such applications.

Building object-oriented interfaces to non-object-oriented programs can be accomplished using the SWIG class extension mechanism. To illustrate, suppose that a object-oriented scripting interface to the ANSI C file I/O operations were to be constructed. This could be done using class extension as follows:

```
%{
#include <stdio.h>
%}
typedef struct { } FILE;
%addmethods FILE {
    FILE(char *filename, char *mode) {
        return fopen(filename,mode);
    }
    int close() {
        fclose(self);
    }
    int flush() {
        return fflush(self);
    }
    int getc() {
        return fgetc(self);
    }
```

```
        int putc(int c) {
            return fputc(c,self);
        }
        int puts(const char *s) {
            return fputs(s,self);
        }
        size_t read(void *ptr, size_t size, size_t nobj) {
            return fread(ptr,size,nobj,self);
        }
        ...
    }
};
```

Using this interface, files could be created and manipulated exactly as if they were objects. For example,

```
# Python script manipulating files
f = File("test","w")          # Open a file
f.puts("Hello world\n")       # Print a message
...
f.close()                     # Close the file
```

The process of building object-based interfaces does not require modifications to the underlying C code nor does it rely on C++. For example, the construction of an object-based interface to the file I/O operations required no changes to the C library nor did it even require the definition of the FILE structure.

5.6 Improving Reliability

Existing programs may experience problems when operating in a scripting environment due to its highly flexible and event driven nature. Although exception handlers can be defined to catch run-time errors, a number of other modifications can be made to improve the stability of scriptable applications.

5.6.1 Execution Order Dependencies

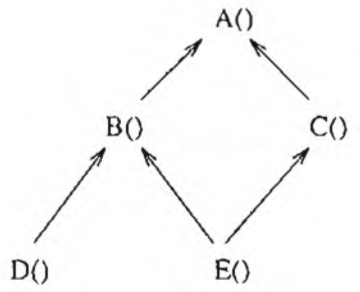
Within a software package, execution order dependencies may implicitly exist between certain functions. That is, assumptions may be made about the order in which functions are to be executed. In addition, certain functions may only be legally executed once while other functions may be used repeatedly. Scripting can cause such applications to fail since functions can now be executed at any time and in any order. Furthermore, when users are unaware of such dependencies and limitations, they can easily write scripts that crash the program or cause erroneous execution.

In Figure 5.2, execution order dependencies between different functions are shown. In the figure, function A must be executed prior to executing functions B or C. Likewise, function D requires the prior execution of B and function E requires the execution of functions B and C. To capture the relationship between these functions, additional state variables can be added to the application and used as a safety check. For example, the original code might be modified as follows:

```
// State variables
A_init = 0;
B_init = 0;
C_init = 0;
// Functions
A() {
  ...
  A_init = 1;
}
B() {
  if (!A_init) return;
  ...
  B_init = 1;
}
C() {
  if (!A_init) return;
  ...
  C_init = 1;
}
D() {
  if (!B_init) return;
  ...
}
E() {
  if ((!B_init) || (!C_init)) return;
  ...
}
```

State variables can also be used to handle cases of reentrancy. For example, if a function can only be executed once, it can be modified as follows:

```
A() {
   if (A_init) return;    // Already executed
   ...
   A_init = 1;
}
```



$$\begin{tikzcd}
 & A() & \\
B() \arrow[ur] & & C() \arrow[ul] \\
D() \arrow[u] & E() \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 5.2. Execution order dependencies

SWIG currently provides no mechanism for explicitly specifying the dependencies between functions. As a result, such modifications are generally made to the original application. Although this requires slight modifications to the original code, it has the benefit of improving the reliability of the original application independently of its scripting interface.

5.6.2 Argument Checking

Other reliability problems can occur if invalid values are passed to certain functions. For example, functions might only work on positive values, non-NULL pointers, and so forth. Although the functions can be modified directly, SWIG typemaps can also be used to impose constraints on function argument values. For example,

```
// SWIG interface with argument checks
%typemap(check) double Positive {
    if ($target <= 0)
       SWIG_exception(SWIG_ValueError,"Expected a positive value");
}

// Make sure all FILE * values are non-NULL
%typemap(check) FILE * {
    if ($target == NULL)
       SWIG_exception(SWIG_ValueError,"Received a NULL pointer");
    }
}
...
double log(double Positive);    // Works only on positive values
int fclose(FILE *);             // FILE * must be non-NULL
...
```

In this case, the value checking code is used wherever arguments of `double Positive` and `FILE *` appear. If an improper value is passed to such functions, a scripting language error will be generated.

5.7 Data Management

Most scientific applications are data intensive. Managing data is important and there are several approaches that are commonly used. First, an application may store data in global variables. With this approach, functions implicitly examine and modify the state of the system. A second approach is to use object-oriented techniques in which various types of objects are used to hold data.

The choice of data model has a surprisingly large impact on the nature of the scripting language interface presented to the user. In some cases, the use of global data may simplify a scripting interface by reducing the number of program parameters that are passed to various functions. For example, a simulation code might maintain a large pool of data corresponding to the current state of a simulation and assume that all functions operate on that data. Likewise, object oriented techniques are most useful when working with large collections of objects. For example, a visualization system might allow a user to manipulate different images and viewpoints. In this case, an object-oriented scripting interface would most closely match the intended use of such a system.

Although existing programs may utilize one or both of these data management models, it is sometimes useful to change the data model when building a scripting interface. Such a change can improve the usability and flexibility of an application without requiring any modifications to its underlying implementation. Furthermore, such changes might allow an old application to be used in new and interesting ways.

To change an application relying on global parameters to an object-oriented model, a container class can be used. This class mirrors the global variables used in the original application. Methods for setting and saving the state of the system are then implemented to copy the values of global variables. Finally, wrappers are created around the original functions so that the state of the system is properly managed. For example,

```
// Global variables in original application
extern double Minx;
extern double Miny;
...

// Class for building an OO interface
```

```
class Data {
private:
   // Mirror the global variables here
   double minx, miny, minz;
   double maxx, maxy, maxz;
   ...
public:
   Data();
  ~Data();
   // Set the global variables
   void set_state() {
        Minx = minx;
        Miny = miny;
        Minz = minz;
        ...
   }
   // Save the global variables
   void save_state() {
        minx = Minx;
        miny = Miny;
        minz = Minz;
        ...
   }
   // Now object-oriented implementations of functions
   void memory(int size) {
        set_state();
        ::memory(size);
        save_state();
   }
   void integrate(int nsteps) {
        set_state();
        ::integrate(nsteps);
        save_state();
   }
   ...
};
```

Likewise, an object oriented interface can be transformed using similar techniques. For example,

```
// Global variable holding current state
View *global_view = 0;

// Set the current state
void set_view(View *v) {
    global_view = v;
}
```

```
// Methods operating on the global view
void rotate_right(double deg) {
    global_view->rotate_right(deg);
}

void rotate_left(double deg) {
    global_view->rotate_left(deg);
}
...
```

It is important to note that these transformations do not affect the original application. Rather, they can be used to change the "appearance" of an application when building its scripting interface. As a result, this allows old applications to be used in new and interesting ways.

5.8 Performance Considerations

The performance characteristics of scripting languages is a major concern for most application developers. The poor performance of interpreters is generally well known although not often quantified. The failure of software projects in which the performance of high-level languages was ignored has also been occasionally described in the literature [45].

The performance of scripting languages has a major impact on the design and implementation of scripting interfaces to scientific systems. This section describes the performance properties of scripting languages and associated design considerations.

5.8.1 The Performance of Scripting Languages

Although scripting languages and interpreters have been gaining in popularity, surprisingly little information is available about their performance characteristics. Users are well-aware that that scripting languages are slower than C, but just how much slower are they?

In a recent paper providing a clock cycle analysis of interpreter performance (including Java, Perl, and Tcl), scripting languages were shown to have a performance approximately 400 to 3700 times slower than C on a simple benchmark [88]. Much of this slowdown is due to the overhead of decoding and dispatching scripting language commands. However, the authors also point out that the performance of interpreters is not easily characterized by any single factor:

> The performance of an interpreter cannot be attributed solely to the frequently executed command dispatch loop. Performance is also linked to (1) the expressiveness of the virtual command set and how effectively these virtual commands are used, (2) the use of native runtime libraries, and (3) the way that the virtual machine names and accesses memory. The "architectural footprint" of an interpreted program is primarily a function of the interpreter itself and not of the programs being interpreted, and that the high-level interpreters behave similarly to large SPECint92 applications such as gcc [88, p. 158].

The performance of scripting languages is largely impacted by the use of compiled code. Thus, even though a simple benchmark application written entirely in the scripting language may run 1000 times slower than C, a script making heavy use of a compiled extension may run much faster. For example, the same article showed that for certain operations such as string splitting, concatenation, and file I/O, scripting languages were only 1.2 to 80 times slower than C [88].

5.8.2 The Performance of Compiled Extensions

When SWIG is used to build a scripting interface, the performance of the underlying C code is largely unaffected. Where performance becomes critical is in the use of the scripting interface. Whenever a function is issued from the interpreter, it must be decoded and dispatched to the appropriate wrapper function. The wrapper function must then convert the function arguments into an appropriate representation before calling the C function. Afterwards, the result must be converted back into scripting and control returned to the interpreter.

The dispatch and decoding process may require several thousand machine instructions [88]. Furthermore, the performance is largely affected by the number of function arguments being passed as well as their type (decoding pointers is more expensive than decoding integers for instance). Therefore, users should assume that the execution of a function issued from a scripting language may require several thousand more machine instructions than would have been required in C.

The overall performance degradation of a compiled function due to a scripting interface is entirely dependent upon the amount of work performed by that function and the performance penalty imposed by the scripting language (which varies widely). Table 5.1 shows the performance penalties that would be incurred for a variety of function execution times and scripting language performance penalties. In the table, the number of native instructions represents the number of machine instructions executed by the C function

Table 5.1. Performance penalties of scripting

	Scripting Instructions		
Native Instructions	100	1000	10000
10	11.0	101.0	1001.0
100	2.0	11.0	101.0
10000	1.01	1.10	2.0
1000000	1.0001	1.001	1.01

while the number scripting instructions represents the number of machine instructions required to decode and dispatch a command as well as execute the code contained in the wrapper function.

As a rule of thumb, C functions should perform an order of magnitude more work than the scripting language in order to achieve a performance penalty less than 10%. Thus, if a scripting language requires 1000 instructions to issue a function call, a C function should execute approximately 10000 instructions to achieve less than a 10% performance penalty. To put this in concrete terms, the function would have to perform approximately the same amount of work as required for a matrix multiplication of two 20×20 matrices. Clearly, the performance penalty skyrockets when functions perform very little work. For example, if a function only performed 10 instructions, the performance penalty incurred in the scripted version could easily be more than a factor of 100.

5.8.3 Designing for Performance

The introduction of scripting to an application will always result in a performance penalty. However, the size of the penalty is largely determined by the design and use of the scripting interface. Generally speaking, the following situations result in few performance penalties when used in a scripting environment.

- Functions involving a significant amount of computation.
- Outer loops of computationally intensive operations.
- Infrequently executed functions.

On the other hand, the following situations tend to result in substantial performance penalties when performed from scripting language.

- Repeated use of functions performing little work.

- Inner loops of computationally intensive operations.

- Fine-grained manipulation of large amounts of data.

Although scientific applications vary widely, computational kernels and numerically intensive operations should be written in C/C++. Meanwhile, most high-level operations and control can be implemented with scripts. This arrangement also reflects the complementary nature of systems and scripting languages where systems languages are used for performance critical operations while scripting languages are used for control and component gluing.

CHAPTER 6

SOFTWARE COMPONENTS

Given an existing application, SWIG can be used to construct a scripting interface. In doing so, the application become more usable and flexible. However, one of the most powerful features of scripting languages is their ability to manage software components. Rather than building a huge monolithic package, applications can be decomposed into collection of scriptable modules. These modules can then be assembled as needed to solve particular problems.

The component approach is attractive for a number of reasons. First, it largely eliminates the tangled control logic found in many scientific programs and allows the subsystems forming a large monolithic package to repackaged as a collection of loosely coupled components. Second, it gives applications a well-organized modular structure that allows new features to be added in a sensible manner (since new features can usually be added as new modules). In addition, components simplify development and maintenance since each module is self-contained and more easily managed than a huge monolithic package. Finally, the component approach makes it easier to combine and use different components in an integrated environment. For example, a simulation program and visualization tool could be turned into components and combined to form an integrated simulation and data analysis system. Such integration would streamline the problem-solving process and could allow scientists to be more productive.

This chapter describes the use of scripting language components, decomposition of large applications into modules, integration of components, and design patterns for assembling component based software.

6.1 Scripting Language Components

Scripting languages support two distinct forms of components. First, a component can be written entirely as a script. When the user wants to use the component, it is simply loaded and interpreted. The second type of component is a compiled extension module. These are the types of modules that are created by SWIG. When creating a compiled extension, the original C/C++ code and wrapper functions are compiled in a shared library. The scripting interpreter can then load shared libraries as needed and execute the wrapper functions they contain.

When SWIG is used to create a scripting interface to an application, that application is effectively transformed into a component. As a component, it can be used and combined with other components to create new applications. As a result, the application really becomes a piece of a much larger programming framework. Although it is impossible to cover all aspects of software frameworks here, articles describing the benefits of frameworks and component-based approaches frequently appear in the literature [36, 1, 69]. Detailed information about using popular component frameworks such as CORBA and COM can be found in a variety of books such as [74, 87]. More formal descriptions of frameworks and software architecture can be found in [90].

6.2 Splitting Applications into Components

Large applications usually contain a variety of subsystems that are interconnected with control code. When SWIG is used to add a scripting interface, much of this control code can be eliminated as described in Chapter 5. As a result, the major subsystems of an application can be exposed and split into libraries as shown in Figure 6.1. The libraries, in turn, can be encapsulated in a collection of scripting language extension modules using SWIG.

When using the component-based application, a user only uses the modules that are

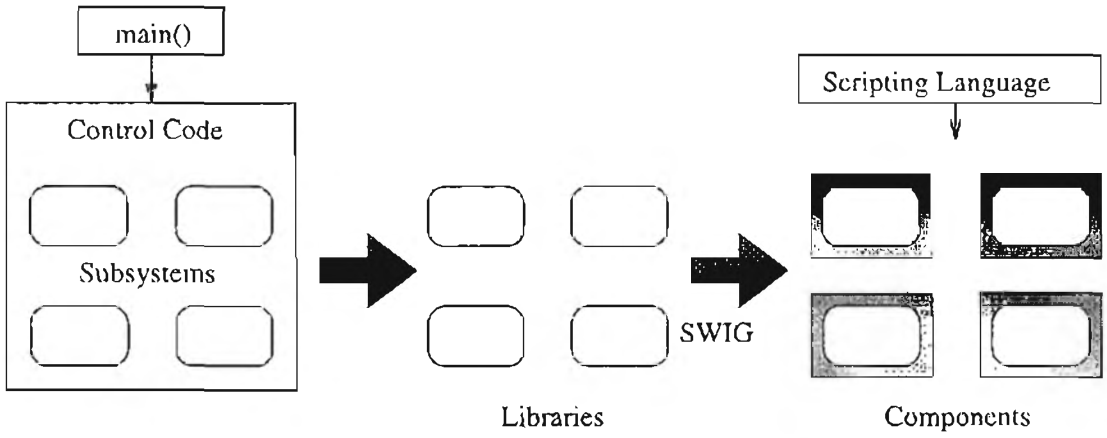


Figure 6.1. Splitting an application into libraries and components

needed to solve the problem at hand. Thus, instead of using a single package containing all of the functionality, problems are solved by assembling an appropriate set of smaller and self-contained modules.

If constructed properly, components can be used from both C/C++ programs and a scripting language environment. Figure 6.2 shows the structure of such a component. In the figure, components are split into two parts: a library and a scripting language wrapper module. SWIG is used to build the wrapper module, but the wrappers are decoupled so that the component can be used as an ordinary link library from other C/C++ programs in addition to being accessible from a scripting interpreter. This arrangement is useful because it provides a strict separation between the implementation of an application and its scripting interface. This allows the components to be used in a wide variety of other settings. For example, a user might use the components, without modification, to create an application entirely in C or C++.

6.3 Systems Integration

One of the biggest problems faced by computational scientists is the decoupled nature of different tools and packages used in the problem solving process. If these tools can be integrated and used together in a more efficient manner, problem solving will be greatly simplified and scientists will be able to use scientific software in a more productive manner.

One way to simplify the use of different tools and packages is to provide a consistent and common interface. Using SWIG, scripting interfaces can constructed for various packages as shown in Figure 6.3. This places different tools under the control of a common scripting language interpreter and makes different systems appear similar. As a result, this tends to simplify the use of these tools and hide implementation differences.

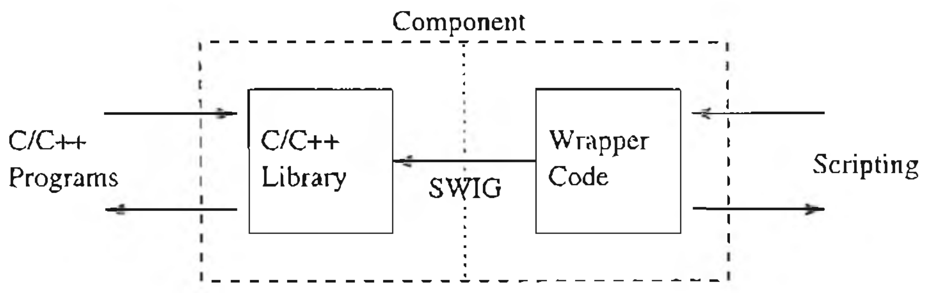


Figure 6.2. Structure of a scripting language component

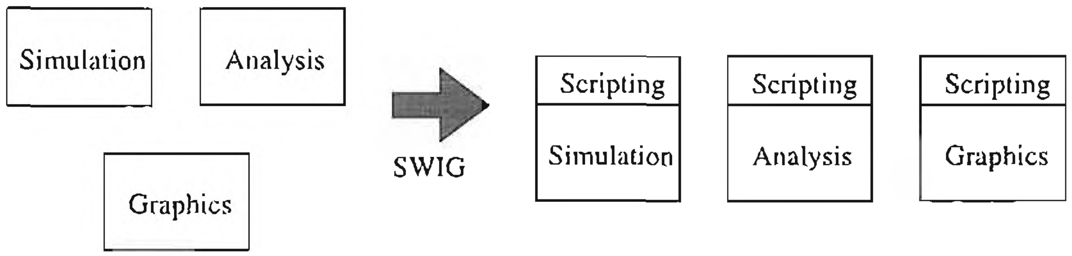


Figure 6.3. Providing a common scripting interface to different packages

Although scripting languages can be used to unify different packages under a common interface, they do not eliminate the decoupled execution of each package. For example, each tool may run independently and exchange data in the form of files and pipes. However, the ability of scripting languages to dynamically load and utilize different software components allows different packages to be integrated directly into a shared process and address space as shown in Figure 6.4. When integrated in this manner, packages can be modified to share data directly and interoperate with each other. This not only improves the efficiency with which different tools can be used, it allows entirely new applications to be developed. For example, the integration of a simulation code and a visualization package could allow a scientist to interactively visualize and steer a running simulation–a process not possible in a decoupled environment.

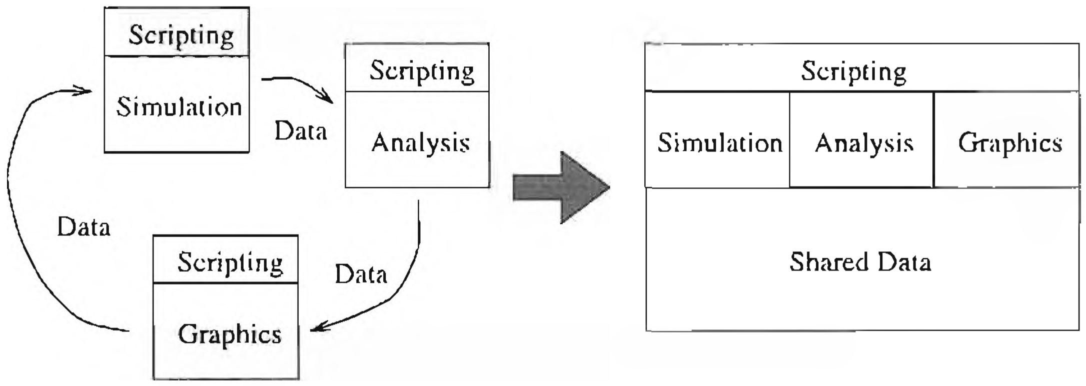


Figure 6.4. Direct integration of packages into into a shared environment

6.4 Component Design

The manner in which components are structured and used is of critical importance. Without special attention, it is possible to create an unmanageable set of components as shown in Figure 6.5. In the figure, there is no discernible structure and each component implicitly depends on the existence of other components. Maintaining such a component framework is difficult since changes to any one component could break large numbers of other components. Likewise, a component may depend on a large number of other components–making such a component extremely sensitive to such changes.

To make a components work effectively, developers need to think about the overall structure of their applications and the creation of components. Ideally, each component should be designed to perform a well-defined set of operations and depend minimally on the use of other modules.

Since software components are similar to objects, many of the object-oriented design patterns and component based software design techniques can be applied to scripted applications built with SWIG [43, 70]. Although an in-depth discussion of object-oriented design techniques is not presented here, there are a few particularly useful types of components that can be used in the construction of a component framework. This section describes these components and the situations in which they can be used.

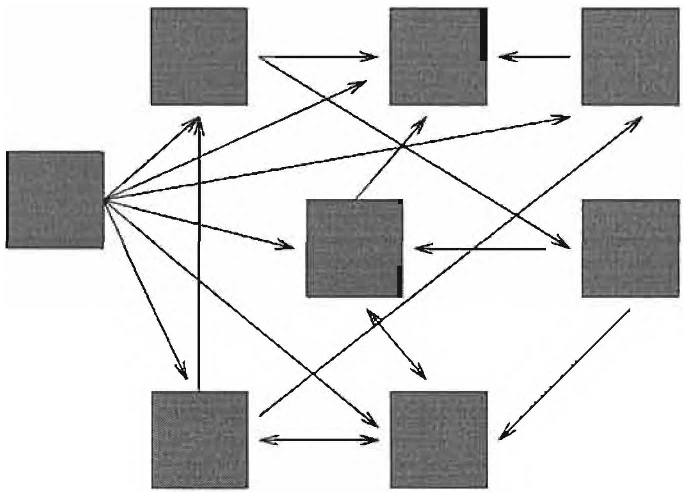


Figure 6.5. A poorly designed set of components

6.4.1 Libraries

The raw functionality of an application or C library can be packaged into a library component. The purpose of a library is to expose functionality to both the scripting language interpreter and other modules written in C. The structure of a library component is shown in Figure 6.6, but was also shown in Figure 6.2. Most of the modules created from an existing application will be libraries.

6.4.2 Adapters

An adapter component (also known as a wrapper) is used to change the interface of an existing component. The primary application of an adapter is to make two previously incompatible components work together. Figure 6.7 shows the structure of an adapter.

Adapters are generally used to add new software components to an existing system without changing any of the existing interfaces. For example, a system might be coded to use OpenGL for graphical display. However, a user might want to port the application to a Windows platform and use Direct3D instead. Rather than changing the application to

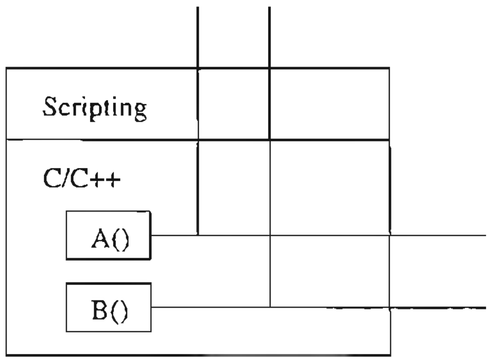


Figure 6.6. A library component

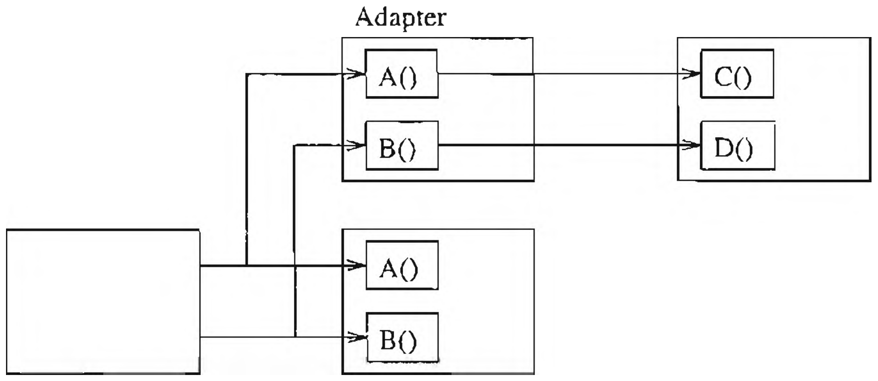


Figure 6.7. An adapter component

use a new interface, an adapter component can be written to give an OpenGL interface to the Direct3D library. Using the adapter, the original application can continue to use the original OpenGL interface even though an entirely different graphics display library is really being used.

The important feature of adapters is that they are used to make a component mimic the behavior of another component in the system. This allows the two components to be used in a more-or-less interchangeable manner.

6.4.3 Bridges

A bridge component is used to provide functionality involving two or more library components in a manner that allows those libraries to remain independent. The structure of a bridge in shown in Figure 6.8.

The purpose of a bridge is to implement functions that require the use of more than one library component, but in a way that preserves the generality of the library components. For example, a system may have general purpose library modules for simulation and graphical display. Using these two libraries, functions for performing integrated data visualization could be implemented. These functions involve both the simulation and graphics libraries, but where should the visualization functions be placed? If they are placed in the simulation library, that library will now depend on the graphics library. Likewise, a similar situation occurs if the functions are placed in the graphics library. The solution to this problem is to place the visualization functions in a separate bridge component. This component provides functionality linking the simulation and graphics libraries together, but in a manner that preserves the independence of those libraries.

6.4.4 Facades

A facade component is used to provide a unified interface to a collection of different components or subsystems. The structure of a facade is shown in Figure 6.9.

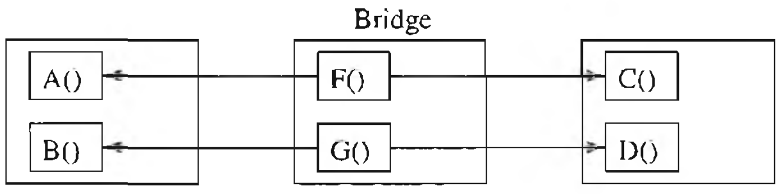


Figure 6.8. A bridge component

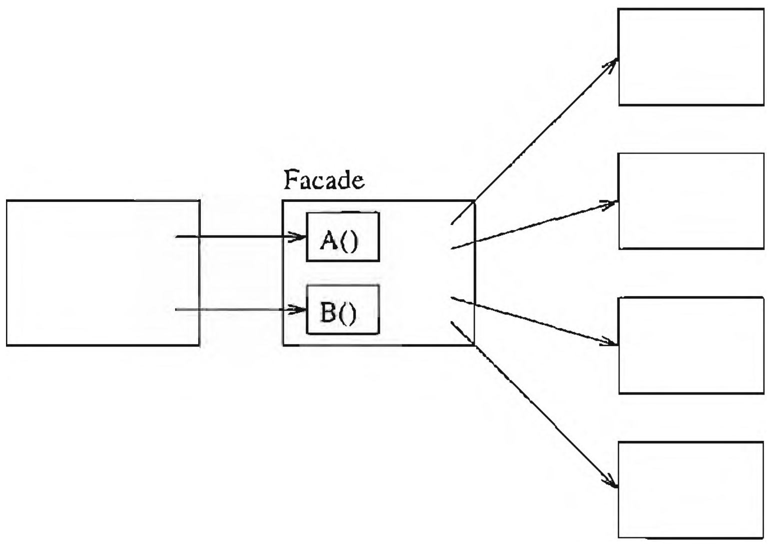


Figure 6.9. A facade component

The purpose of a facade component is to simplify the interface to a variety of subsystems and modules. A facade can also reduce interface complexity and decrease the number of dependencies between components. For example, suppose that a scientific system was to support a collection of different components for making data plots. Further suppose that a variety of different plotting libraries were to be used and that each library had a different interface. Rather than writing code that interfaced to every possible plotting library, a plotting facade component could be developed. The facade would provide a generalized plotting interface that clients could use to make plots using any of the plotting components available in the system.

The benefits of facades are that they shield clients from the subsystem components and make these components easier to use. Facades also allow the individual subsystem components to be changed and updated independently without affecting any of the clients (i.e., weak-coupling).

Facades are similar to adapters but are used in a slightly different way. An adapter is typically used to make the interface of a component mimic that of another component. A facade, on the other hand, is used to create a generalized interface to a collection of different components.

6.4.5 Building a Component Library

When creating component libraries, the use of library, adapter, bridge, and facade components can greatly simplify the organization and maintainability of the library by reducing the number of dependencies between components and providing structure. Figure 6.10 shows a well-structured library of components in which the types of components previously described have been utilized. In this case, the number of dependencies has been greatly reduced. As a result, this collection of components would be easier to develop, maintain, and use than those shown in Figure 6.5.

6.5 SWIG and Component Building

SWIG only plays an indirect role in the construction of components and component-based applications. By simplifying the construction of scripting interfaces, SWIG allows existing applications to be easily incorporated into a scripting environment. The strong component flavor of scripting languages then encourages developers to think about breaking applications into components and combining different systems within a component framework.

When building scripting components, SWIG enforces no particular design philosophy on users. As a result, it is largely up to the user to create a sensible mechanism for building and hooking different components together.

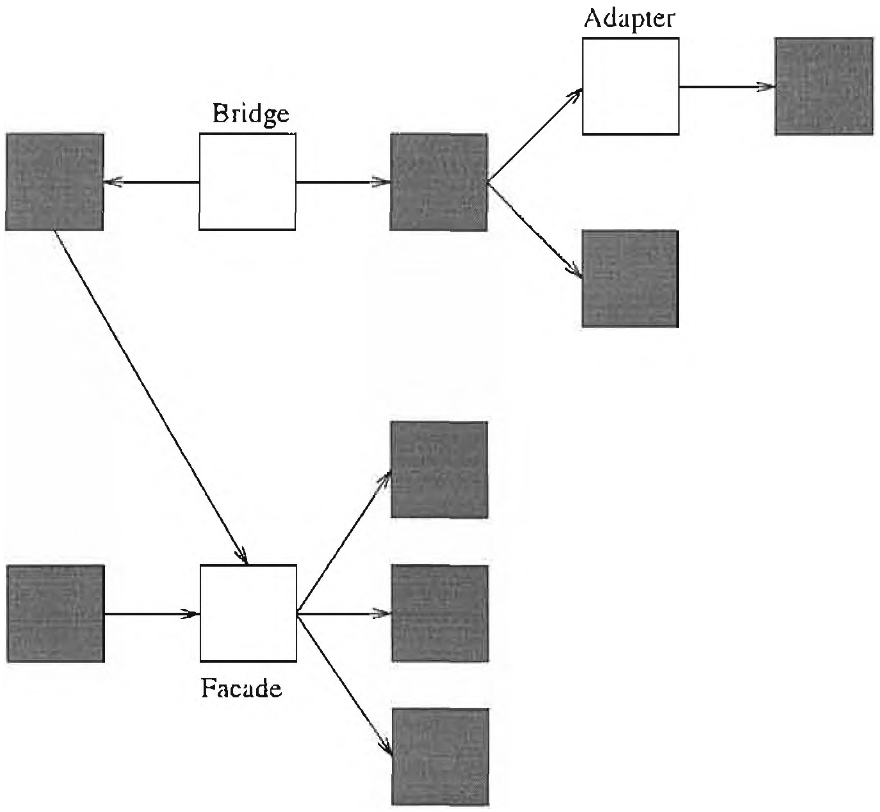


Figure 6.10. A designed component library

CHAPTER 7

CASE STUDY : MOLECULAR DYNAMICS

SWIG was originally developed for use with the SPaSM molecular dynamics code at Los Alamos National Laboratory. This chapter describes the use of SWIG and scripting languages with this application over a 3-year period from 1995 to 1998. Particular attention is given to evolutionary changes made to improve the usability and structure of this application. The goal of this chapter is to present a case study describing how SWIG has been applied to an existing application, how that application has improved over time, and what the impacts of such improvements are on the problem-solving process.

7.1 The SPaSM Code

SPaSM (Scalable Parallel Short-range Molecular dynamics) is a simulation code developed at Los Alamos National Laboratory for performing large scale three-dimensional simulations of materials using the method of molecular dynamics [10, 3]. Applications include crack propagation, dislocation dynamics, friction, and shock waves [110, 111, 57, 58].

SPaSM was originally developed in 1992 for the Connection Machine 5 massively parallel supercomputer [10]. It has since been ported to a variety of parallel machines including the Cray T3D, IBM SP-2, SGI Origin 2000, and Sun Enterprise servers. It also operates on single processor workstations and clusters. In 1993, SPaSM was one of the winners in the 1993 Gordon Bell Prize competition for achieving 50 Gflops performance on the 1024 processor CM-5 at Los Alamos National Laboratory [64]. SPaSM is implemented entirely in ANSI C with explicit message passing used for interprocessor communication. A multithreaded version of the code for running on shared memory systems is also available.

Prior to 1992, most molecular dynamics simulations were limited to a few hundred thousand atoms and mostly performed in two dimensions [14]. SPaSM was the first molecular dynamics code to perform a simulation with more than 100 million atoms in three dimensions and has since been used to perform a variety of production simulations involving millions to tens of millions of atoms.

Development of SPaSM has been an ongoing process. Much of the development has involved the introduction of new physical models and the creation of code used to study different physical problems. A scripting language was added to SPaSM code in 1995 and is being used as a foundation for making further improvements today.

7.2 Before SWIG

In Chapter 2, problems related to scientific software such as piecemeal growth and user interfaces were discussed. This section describes these issues with respect to SPaSM.

7.2.1 Development of SPaSM

The first version of SPaSM was written to perform simple three-dimensional molecular dynamics simulations using a short-range Lennard Jones interatomic potential [10]. The primary goal was to develop and test parallel algorithms for short-range molecular dynamics and to investigate the scalability and performance of these algorithms. The initial implementation, consisted of approximately 3000 lines of ANSI C and could perform simulations with as many as 67 million atoms.

In 1993, development primarily focused on achieving better performance. Although good scalability was observed, SPaSM achieved only 1.5% of the peak performance of the CM-5. To improve performance, assembly code was written to drive the CM-5 vector units which resulted in a factor ten performance improvement. A few additional features including new boundary conditions and table lookup methods were also added to the code at this time. By the end of 1993, the code had doubled in size to approximately 6000 lines and could perform simulations with as many as 180 million atoms [15].

By late 1994, SPaSM had been ported to a variety of other parallel machines. In addition, development had focused on making the code better suited to production computing. Capabilities for I/O, checkpointing, and restarting were improved and new physical models were introduced. A memory optimization also allowed for simulations with as many as 300 million atoms. At this time, a number of other scientists had started using and expanding the code which had now more than doubled in size to 17000 lines.

By mid 1995, SPaSM had grown to nearly 25,000 lines. The number of users had also increased, making software maintenance problematic. Not only were there many different configurations, modules, and extensions, users would often copy the source and make changes to their local copy. Incorporating these changes back into the master version was extremely difficult since it was common for each user to have slightly modified versions of the code that was incompatible with all of the other versions for one reason or another. By now, it was clear that an alternative approach for organizing and using the system would be useful.

7.2.2 User Interfaces

SPaSM was originally controlled through the use of command line options and interactive input. A sample session appears as follows:

```
% SPaSM -i1 -p8:8:8 -m100000 -c4:4:8 -r0:0:0:80:80:160 -t0.001 \
  -e10 -o100
Starting Run 17.
Initializing Node Processors...
Setting up initial conditions...
Number of Particles : 1024000
Nsteps : 1000
Integrating 1000 timesteps...
Nsteps : 1000
Integrating 1000 timesteps...
Nsteps : 0
Writing data to Save17
%
```

To run the code, all of the simulation parameters were specified as command line options. The user was then queried for the number of integration timesteps. Entering a positive number would result in that many numerical integration steps. Entering zero or a negative number would force the program to exit.

As more features were added, the use of command line options became difficult to manage. Not only was it difficult for users to remember all of the command line options, it was clear that a more flexible user interface would be needed for continued development.

A rudimentary command interpreter was added in 1994. This interpreter allowed the user to interactively enter a keyword followed by a value. The keywords corresponded to C global variables that could be queried or modified. In addition, certain keywords would trigger internal functions. With this interface, the code was controlled through both command line options and the interpreter. Although this approach made it somewhat easier to change parameters and execute simple functions, the interpreter was quite limited in functionality. To complicate matters, adding new parameters to the interpreter required users to write C code such as the following:

```
void
initcond_command(char **tokens)
{
 char *Commands[NC], *Format[NC], *Carg[NC];
 PFI  Cptr[NC];
 char  **cp, **fp, **ap;
 PFI   *pp;

 cp = Commands; fp = Format;  ap = Carg;  pp = Cptr;
 *(cp++) = "aspectx";    *(fp++) = "%d";
 *(ap++) = (char *) &aspectx;  *(pp++) = NULL;
 *(cp++) = "aspecty";    *(fp++) = "%d";
 *(ap++) = (char *) &aspecty;  *(pp++) = NULL;
 *(cp++) = "aspectz";    *(fp++) = "%d";
 *(ap++) = (char *) &aspectz;  *(pp++) = NULL;
 ...
 *(cp)   = NULL;
 /* Parse commands */
 parse_commands(tokens, Commands, Format, Carg, Cptr);
 ...
 }
}
```

By 1995, it was clear that this user interface scheme was not going to scale as the application continued to grow. Furthermore, there was a growing interest in adding data analysis and visualization capabilities to the system. These additions would be significantly more complicated than anything that had previously been written. Thus, a better scheme for controlling the application needed to be devised.

7.2.3 Data Analysis and Visualization Woes

The primary goal of SPaSM was to investigate molecular dynamics simulations on a scale not previously possible. Even though such simulations could be performed, they would typically take tens to hundreds of hours of CPU time to complete and were always submitted as batch processing jobs after a suitable set of simulation parameters were determined. It was not uncommon to have tens of gigabytes of output data to analyze after each simulation. To analyze data, each datafile would be transferred over the network to a local workstation. Using a standard 10 Mbps network connection, the transfer of a 1.6 Gbyte datafile would take between 30 minutes to an hour depending on network load. Once available locally, the datafile would be fed into a visualization tool. Most visualization was performed using a customized visualization tool written for a high-end SGI Onyx workstation. Although this tool theoretically allowed the user to interact with the data, it often required several hours to render a single image and was usuable for large datasets.

Although large simulations could be performed with SPaSM, analysis and visualization of those simulations proved to be a painful process involving days and even weeks of effort. Even for small simulations, the process was far from easy. For example, one simulation involving 1.2 million particles resulted in 1000 datafiles each about 20 Mbytes in size (20 Gbytes of data). Although the entire simulation required less than 6 hours of CPU time to run, visualizing the data to produce an animated movie of the time evolution required more than a week of continuous processing on two high-end raphics workstations.

From a usability standpoint, this situation was unacceptable. Scientific computing is an inherently exploratory activity. Yet, the vast amounts of data made such exploration virtually impossible. In fact, the process was so difficult, the first 3 years of the SPaSM project saw only a handful of "real" simulations.

7.2.4 The Need for a New Approach

After three years of frustration, a new approach had to be developed to make large-scale molecular dynamics modeling practical. The data analysis and visualization problems were the greatest concern, but better approaches for controlling and managing the simulation code were also needed.

The primary obstacle to effective data analysis was the decoupled nature of simulation, analysis, and visualization. When these tasks are decoupled, analysis is performed by taking the output files of a simulation and feeding them into an analysis package (often located on a different machine). This package might generate additional data files that could be used for visualization and so forth. Unfortunately, for large scale molecular dynamics, the amount of data easily overwhelms existing tools and makes this approach highly ineffective (for example, loading a large MD dataset into AVS would cause the system to crash).

An obvious solution to the data analysis problem is to provide better integration between simulation, analysis, and visualization tools. Rather than performing these tasks separately on different machines, perhaps everything could be performed on the high performance supercomputing system. Furthermore, research efforts in computational steering had demonstrated that the integration of these tasks greatly improved the usability of scientific systems [80, 49]. Given the data analysis problems with SPaSM, it was decided that a steering approach might provide the greatest benefit to users. With such an approach, tasks that were decoupled would be integrated. This integration would eliminate the need to transfer huge amounts of data between tools and systems. This, in turn, would allow users to perform more simulations and to be more productive.

Although this idea is highly attractive, it is also problematic. How would such a system be assembled? How would a user control it? How would it be extended with new functionality? At the time, SPaSM was structured in a relatively ad-hoc manner and controlled through a weak user interface. Adding such a sophisticated data analysis capability would certainly require a more powerful approach for controlling and building scientific software.

7.3 The SWIG Prototype

In 1995, a prototype scripting system was constructed for the SPaSM code [11]. This system consisted of a simple scripting language and an automatic code generator for building extensions. This section describes the implementation, use, and results of using the prototype. Many of the lessons learned in this stage went into the development of the SWIG compiler and future versions of SPaSM.

7.3.1 A Scripting Language and Compiler

Due to the special purpose nature of parallel machines and difficulty of using existing software, a simple parallel scripting language was implemented using Lex and Yacc [62]. This scripting language supported a few useful datatypes, provided all of the constructs found in a normal computer language (procedures, loops, conditionals, variables, etc...), and could operate properly on parallel machines (mainly an issue of proper I/O handling). This language could also be interfaced to C functions. Thus, the idea was that all of the C functions in SPaSM could be exposed to the user as "commands" in this language. In addition, the command driven interface could be used to interactively drive data analysis and visualization features when they were eventually added to the system.

To provide access to C functions, the scripting language required wrapper functions like other scripting languages. To generate these functions automatically, a simple compiler was developed to turn simple ANSI C declarations into wrapper code. This compiler only supported global variables and functions. In addition, it only supported four C datatypes (`int`, `double`, `char *`, and `void`). Although limited, this was enough to support most of the SPaSM code.

7.3.2 Building the Initial System

With the scripting language and wrapper code compiler in place, a scripting interface to SPaSM was constructed by copying C header files and editing them slightly to create an interface file for the wrapper generator. Most C functions in the system were already defined in header files so this process effectively exposed most of the underlying functionality to the scripting interface. The wrapper code compiler made the process of building the scripting interface surprisingly easy. In fact, the initial interface description was created in approximately 15 minutes.

Since the scripting language replaced the old command interpreter and command line options, the SPaSM `main()` function was modified to initialize and pass control to the scripting language interpreter upon startup. Other than modifying `main()` and adding the scripting interpreter, no other parts of the SPaSM code were modified. Given that the system had grown to approximately 25,000 lines of code, modifying the system in a drastic manner was not an option since it was unclear whether the scripting approach would actually work. Therefore, changes were initially kept to a minimum.

7.3.3 Using the Scripted Version

When running the scripted version of SPaSM, the user was presented with a command prompt. At the command prompt, the user could execute various C functions, set variables, query parameters, and execute scripts such as the following:

```
! Script for strain-rate experiment
! Run parameters
exdot = 0.0;
eydot = 0.001;
ezdot = 0.0;
lx  = 80;
ly  = 40;
lz  = 10;
lc  = 20;
gapx = 5.0;
gapy = 25.0;
gapz = 5.0;

! Set up a morse potential
alpha = 7;
cutoff = 1.7;
source("Examples/morse.script");
makemorse(alpha,cutoff,1000);  ! Create a morse table
init_table_pair();             ! Use a tabulated pair-potential
```

```
! If restarting from a file, the variable Restart is set to 1
! We'll only create the initial condition if not restarting
if (Restart == 0)
   ic_crack(lx,ly,lz,lc, gapx, gapy, gapz, alpha, cutoff);
   set_initial_strain(0,0.017,0);
endif;

! Now set up the boundary conditions
set_strainrate(exdot,eydot,ezdot);
set_boundary_expand();

Benchmark=1;        ! Report timing information
MovieMode=1;        ! Output frame numbers for visualization
FilePath="/sda/sda2/beazley/test";
output_addtype("pe");           ! Output potential energy

! Run it
timesteps(1000,10,50,500)
```

In the script, most of the function calls correspond to C functions in the original application. Capitalized variables such as `Benchmark` are mapped directly onto C global variables used to hold various run-time parameters (by convention, global variables were capitalized in SPaSM). Most other variables are local to the scripting interpreter and not visible to the underlying C implementation.

Although this approach was somewhat similar to earlier interface schemes, it was also significantly more powerful. Rather than creating simple input files, users could write sophisticated simulation scripts. These scripts could contain control logic and even define new functions. As a result, the scripting language interface was much more than a simple user interface-in fact, it effectively became an interpreted extension of the underlying C code.

7.3.4 Dead Code Elimination

By adding a scripting interface, the old user interface scheme was rendered obsolete. However, significant portions of code related to the old user interface were still in place despite being inoperative (or dead). With scripting successfully in place, this code (corresponding to about a thousand lines of source) was gradually eliminated. As a result, SPaSM was transformed primarily into a large library of functions and variables. Even though much of the control logic linking various functions together remained in place, the code was gradually simplified as it was freed from its original user interface.

7.3.5 Improving Reliability

Prior to the use of scripting, SPaSM required a precise sequence of operations when running a simulation.

1. Specify problem parameters.
2. Initialize particle memory.
3. Initialize the problem geometry.
4. Initialize the force calculation.
5. Set up an initial condition.
6. Run the simulation.

The sequence in which these operations were performed was hard-coded into the original implementation and many of the steps assumed the successful completion of earlier steps. For example, in order to create an initial condition, it was assumed that particle memory and local geometry had been initialized.

To properly handle these execution order dependencies, SPaSM was modified with state variables using the same process described in Chapter 5. For example

```
int Memory_Init = 0;
int Geometry_Init = 0;

void memory() {
    ...
    Memory_Init = 1;
    return;
}

void geometry() {
    if (!Memory_Init) {
       printf("Memory not initialized!\n");
       return;
    }
    ...
    Geometry_Init = 1;
    return;
}
```

```
void initcond() {
    if (!Geometry_Init) {
       printf("No geometry initialized!\n");
       return;
    }
    ...
}
```

Although the required modifications were minor and simple to implement, such changes greatly improved the reliability of the code and made it difficult for the user to crash the system by issuing commands in the wrong order.

A related problem was that of function "reentrancy." Previously, certain functions could only be used once during a simulation. However, with scripting, it became possible for the user to repeatedly call functions with new parameters. Many functions in SPaSM were unprepared for this possibility and would cause a crash if this occurred. To fix this problem, many functions were modified to check for prior use as follows:

```
int Memory_Init = 0;
...

void memory() {
    if (Memory_Init) {
       // Memory already initialized.
       // Reallocate memory
       ...
    } else {
       // Initialize new memory
       ...
    }
    ...
    Memory_Init = 1;
    return
}
```

In the months following the introduction of a scripting interface, most of the critical C functions in SPaSM were modified slightly to check for proper initialization and reentrancy. These changes, although minor, greatly improved the stability of the system and made it difficult for users to inadvertently crash the code. It also allowed SPaSM to be used in new ways. By making functions reentrant, it was possible to completely reconfigure a simulation on the fly. For example, a user could dynamically change the partitioning of data across processors or change the simulation geometry by simply issuing appropriate commands. Previously, such changes would have required checkpointing and restarting the entire simulation.

7.3.6 Integrated Data Analysis and Visualization

To add a data analysis and visualization capability, a simple parallel graphics library was implemented [11]. In addition, a number of application specific visualization functions were written. Previously, these functions were contained in a separate package for use on an SGI workstation. However, in the new implementation, the functions were written to directly examine the molecular dynamics data in memory and use the graphics library to create various types of plots.

As output, the graphics system produced GIF images [72]. These images were sent across a socket connection to a server running on the user's workstation [93]. When the server received an image, it was displayed using an image display tool such as `xv`.

To control the analysis and visualization system, the C functions forming the system were wrapped and included in the scripting language interface. By typing various commands interactively, the user could then create and manipulate images as follows:

```
SPaSM [1] > open_socket("sol.cs.utah.edu",32487)
Opened connection with sol.cs.utah.edu
SPaSM [1] > imagesize(500,500);
SPaSM [1] > colormap("cm15");
SPaSM [1] > range("ke",0,10);
SPaSM [1] > image();
SPaSM [1] > rotr(45);
SPaSM [1] > rotd(10);
SPaSM [1] > zoom(200);
SPaSM [1] > down(25);
SPaSM [1] > clipx(45,55);
...
```

The introduction of an integrated visualization capability revolutionized the use of SPaSM. Since visualization and analysis were now part of the simulation package, they could be performed at any time during a simulation. Furthermore, these tasks could be accomplished without using a separate tool or transferring large datafiles between machines. Using the visualization capability, datasets as large as 100 million atoms could be visualized and displayed on an ordinary workstation in as little as 15 seconds over a standard T1 internet connection. By comparison, similar tasks using the older tools used to require tens of hours.

7.3.7 Lessons Learned

The prototype scripting system was used with SPaSM for approximately one year and revolutionized the manner in which simulations were performed. The use of scripting also had an impact on the implementation and underlying structure of the software.

Scripting languages. The prototype demonstrated that scripting languages could be used to drive high performance scientific applications. In fact, scripting proved to be superior to any of the interface techniques that had previously been used.

Automatic wrapper code generation. An automatic wrapper code generation tool proved to a highly effective method for building scripting language interfaces. This tool allowed the original application to be scripted in a relatively short amount of time. It also allowed developers to focus on other aspects of the system and the addition of new functionality. In fact, in the year that the prototype was used, no wrapper functions were written by hand, nor were any errors attributed to the use of an automated tool.

Improved software. The introduction of scripting resulted in improved software. Since scripting replaced the original user interface, that code could be stripped from the application and thrown away. In addition, the event-driven nature of scripting resulted in a number of minor modifications to make SPaSM more robust and usable.

Benefits of integration. The integration of simulation, data analysis, and visualization revolutionized the use of the code. Tasks that used to take hours could now be performed in seconds, while tasks taking days could be performed in a matter of minutes. This not only made the code more usable, but made it practical for scientists to perform and analyze large-scale molecular dynamics simulations on an everyday basis.

Evolutionary improvement. At no time during its transformation was the SPaSM code inoperable. The wrapper code compiler could be used to add scripting to SPaSM with few modifications. As minor changes were made and the code improved, the scripting interface was easily evolved and maintained.

7.3.8 Limitations

Even though the prototype was highly successful, it also suffered from a number of drawbacks.

Choice of scripting language The prototype utilized a custom scripting language. Although this language was functional, it was severely limited. There were no high-level data structures such as lists and associative arrays, no error handling mechanism, nor any support for object oriented programming. To further complicate matters, there was no documentation and support for users. Compared to other scripting languages, it was clear that bringing the custom scripting language up to a comparable standard would involve a substantial and long-term development effort and that such an effort would be of questionable value.

Limitations in wrapper generation The automatic wrapper generator provided only four C datatypes and only supported functions and global variables. Although this proved to be sufficient for building a working prototype, better results might be achieved though a more sophisticated compiler capable of supporting a larger subset of the C language.

System organization Although scripting had proven to be an effective means for controlling simulations and visualization, little attention was given to the overall structure of the application. In fact, SPaSM was still a large monolithic package where all of the different pieces of the system were linked together and combined to create a large executable. In addition, much of the control logic found in earlier implementations remained in effect despite the improved interface.

Usability The new system was significantly more usable than prior versions, but was by no means perfect. The relatively unstructured nature of the original application resulted in a scripting interface with minor inconsistencies and quirks. Most of the usability problems were due to inexperience with scripting language interfaces and limitations in the design of the original SPaSM implementation. Thus, even though it was relatively easy to add a scripting interface, that interface was not without problems.

7.4 SWIG and Python

Many of the lessons learned in the prototype went into the development of SWIG. In 1996, the prototype was replaced with a new scripting interface based on Python and SWIG [7]. This section describes that transition and additional improvements made to the code.

7.4.1 Building a Python Interface

Python was selected as a replacement for the prototype scripting language [66]. Python is a freely available object-oriented scripting language that has been used increasingly in scientific applications. Since the scripting language interface in the prototype was generated automatically with a precursor to SWIG, changing SPaSM to use Python was a simple process. Interface files from the prototype were modified slightly and fed into the SWIG compiler to generate Python wrappers. In addition, the `main()` function was modified to initialize and start the Python interpreter. No other changes were made to the source code.

The conversion process required a few hours of work, but most of this time was spent modifying Makefiles and other parts of the build process to use Python instead of the older scripting interface. After the conversion, SPaSM operated in an identical manner as before except that the scripting interface was now a Python interpreter. Aside from a few minor syntax changes, scripts developed for the older scripting language could be easily adapted to Python and used exactly as before.

7.4.2 Splitting SPaSM into C Libraries

Since Python supports dynamic loading of modules, SPaSM was restructured as a collection of components. To do this, major parts of the system were identified, isolated, and turned into the following collection of C libraries:

SPaSM library. All molecular dynamics simulations rely upon a core set of algorithms and functions. This library contains general purpose functions for memory management, parallel MD algorithms, data distribution routines, I/O functions, and so forth. The contents of this library almost exactly match the first version of code developed in 1992 (although almost all of the algorithms have been refined and improved).

System library. To achieve portability across a wide range of platforms, SPaSM is implemented over a machine-independent collection of message passing and threading wrappers. This library provides the implementation of these wrappers and is used as a facade component by most other modules.

Graphics library. The parallel graphics library used for the visualization system consists primarily of functions for creating images and primitives for two-dimensional and three-dimensional graphics. The graphics library is entirely general purpose and does not rely upon any of the data structures or functions contained within other modules.

Analysis and visualization library. The analysis and visualization library acts as a bridge between the SPaSM code and the graphics library. Although the graphics library is generic, this library directly accesses particle data to create and display images.

Simulation libraries. Specific scientific problems involve a certain amount of customized code such as physical models, boundary conditions, numerical integrators, and so forth. The implementation of these features may be different for each problem that is solved. Therefore, each problem is encapsulated into a separate library. For example, functions used to solve shock wave problems are packaged into a separate library than functions used to study dislocation crossings. These libraries generally live in user directories as opposed to being part of the generic SPaSM implementation.

By creating these libraries, SPaSM was split into logically distinct pieces. In the process, much of the control logic holding the system together was removed or rewritten. Even though few changes were made to the underlying algorithms, significant changes were made to the overall structure of the application. Most of these changes were made to the interfaces between libraries in order to make each library self-contained and generic.

7.4.3 Creation of Python Modules

With SPaSM split into C libraries, Python modules were created to provide access to these libraries. The C header files describing each library were modified slightly to serve as SWIG interface files. The headers could then be given to SWIG to create Python wrappers. These wrappers were compiled and linked against the C libraries to form a collection of dynamically loadable Python extension modules.

Although there is a one-to-one mapping between Python modules and C libraries, there is a strict separation between the C implementation and Python interface as described in Chapter 6. The C libraries do not contain any Python specific code and can be used to create stand-alone C executables. Likewise, the Python extension libraries only contain the wrapper code needed to build the Python interface, but these are merely linked against the C libraries to create the full extension module.

Finally, to simplify the maintenance and use of the components, they were placed in a common repository accessible to all users. One of the biggest problems with the prototype was that users made copies of the source code and started working with a local copy. This made maintenance and development difficult since each user always ended up with a different copy of the code. With a component repository, a common collection of components could be given to all of the users though access to a centralized component library. When changes were made to each component, those changes would automatically be propagated to all of the other users.

7.4.4 Object-Oriented Extensions

Unlike the prototype wrapper generator, SWIG allows scripting interfaces to C data structures to be built. To exploit this, the scripting interface was modified to include various data structures within the SPaSM code. For example, particles are described by the following data structures:

```
typedef struct {
  double x,y,z;
} Vector;

typedef struct {
  int    type;
  int    tag;
  Vector r;
  Vector s;
  Vector f;
  double pe;
} Particle;
```

When given to SWIG, these structures are converted into methods for accessing and manipulating `Particle` objects from the Python interface. Thus, if p is a particle object, a user can issues commands such as the following:

```
SPaSM [39] > print p.type
2
SPaSM [39] > print p.r.x, p.r.y, p.r.z
0.870868933099 0.1 11.5630339965
SPaSM [39] > p.tag = 1
SPaSM [39] >
```

To improve the interface to data structures, the SWIG class extension mechanism was used to attach "methods" to C data structures. For example, the following interface file attaches methods for extracting particles, output, and array indexing.

```
// Have SWIG attach the following methods to Particles
%addmethods Particle {
    /* Return a pointer to the nth particle */
    Particle(int n) {
       return ((Particle *) Particles) + n;
    }
    /* Array indexing method */
    Particle *__getitem__(int n) {
       return self+n;
    }
    /* Create a string representation of a particle */
    char *__str__() {
        static char s[1024];
        sprintf(s,"type : %d\n\
tag  : %d\n\
r    : [%0.17g, %0.17g, %0.17g]\n\
s    : [%0.17g, %0.17g, %0.17g]\n\
f    : [%0.17g, %0.17g, %0.17g]\n\
pe   : %0.17f\n",
         self->type, self->tag,
         self->r.x,self->r.y,self->r.z,
         self->s.x,self->s.y,self->s.z,
         self->f.x,self->f.y,self->f.z,
         self->pe);
    return s;
    }
}
```

The added methods do not change the internal C implementation of particles, but they make it possible for the user to manipulate and view particles as follows:

```
SPaSM [39] > p = Particle(10)      # Get the 10th particle
SPaSM [39] > print p
type : 0
```

```
tag  : 0
r    : [0.10000000000000, 1.6417378661970, 12.333902929576]
s    : [0.02565919519959, 0.1543604111549, 8.4869030486381]
f    : [0, 0, 0]
pe   : -5.972883777266366

SPaSM [39] > # Find the particle with the highest pe
SPaSM [39] > p = Particle(0)
SPaSM [39] > max = -9999999
SPaSM [39] > for i in xrange(0,SPaSM_count_particles()):
...             if p[i].pe > max:
...                     pmax = p[i]
...                     max = p[i].pe
...
SPaSM [39] > print pmax
type : 0
tag  : 0
r    : [0.87086893309851, 21.684330126758, 127.19337396125]
s    : [0.04636369788149, -0.0421596288401, 0.08774536227903]
f    : [0, 0, 0]
pe   : -4.29142941533476296

SPaSM [39] >
```

The ability of SWIG to extend structures into classes also proved to be useful in the graphics system. To hold image information, a C data structure Image was used. Various graphics operations then required a pointer to an Image structure as follows:

```
Image *create_image(int width, int height);
void   destroy_image(Image *img);
void   plot(Image *img, int x, int y, int color);
void   line(Image *img, int x1, int y1, int x2, int y2, int color);
...
```

With SWIG, this functionality could be repackaged and attached directly to the Image data structure as follows:

```
%addmethods Image {
    Image(int w, int h) {
          return create_image(w,h);
    }
   ~Image() {
          destroy_image(self);
    }
    void plot(int x, int y, int color) {
          plot(self,x,y,color);
    }
```

```
        void line(int x1, int y1, int x2, int y2, int color) {
              line(self,x1,y1,x2,y2,color);
        }
        ...
    }
```

Within the Python interface, images now operate as if they were defined by a C++ class. For example,

```
    SPaSM [39] > i = Image(400,400)
    SPaSM [39] > i.plot(200,200,1)
    SPaSM [39] > i.line(10,10,395,150,2)
    SPaSM [39] > del i
```

7.4.5 Exception Handling

To further improve the reliability of the code, an exception handling mechanism was added to many of the libraries. C macros for "Try" and "Except" were implemented using the C <setjmp.h> library. Various C functions were then modified to throw exceptions such as follows:

```
    /* A C function that throws an exception */
    void *SPaSM_malloc(size_t nbytes) {
        void *ptr = (void *) malloc(nbytes);
        if (!ptr) Throw("SPaSM_malloc : Out of memory!");
        return ptr;
    }
```

If an uncaught exception occurs, the C program prints a message and exits. However, other C functions were modified to catch exceptions and recover if possible. For example,

```
    /* A C function catching an exception */
    int foo() {
        void *p;
        Try {
            p = SPaSM_malloc(NBYTES);
        } Except {
            printf("Unable to allocate memory. Returning!\n");
            return -1;
        }
    }
```

The exception handling mechanism was also hooked into the Python exception handler using SWIG. This was done by defining an exception handler as follows:

```
    // A user defined exception handler
```

```
%except(python) {
     Try {
          $function
     } Except {
          PyErr_SetString(PyExc_RuntimeError,SPaSM_error_msg());
     }
}
// C declarations
...
```

The handler code gets placed into all of the Python wrapper functions and effectively translates C exceptions into Python exceptions. Now, when SPaSM is used from Python, exceptions in the C code simply result in Python errors as follows:

```
SPaSM [39] > SPaSM_memory(50000000)
RuntimeError: SPaSM_malloc(505032704). Out of memory!
(Line 52 in memory.c)
SPaSM [39] >
```

7.5 The Current Implementation

The current version of SPaSM makes extensive use of SWIG and its Python interface. This section briefly describes the organization and use of the system.

7.5.1 Components

As described previously, SPaSM was split into a collection of C libraries that form the core set of components of the system. Associated with each library is a Python wrapper extension module that exposes the functionality of each library to the user. In addition to the core libraries, a number of modules are enhanced through additional code written entirely in Python. For example, the visualization system is now implemented partially in C and Python. Figure 7.1 shows the overall organization of the system and Table 7.1 shows the implementation details of each component.

Unlike the monolithic nature of earlier versions, SPaSM is now maintained entirely as a collection of components. Attempts to minimize the dependencies between components has also allowed most modules to be maintained separately. A number of components also function as facades and bridges as described in Chapter 6. For example, the analysis and visualization library serves as a bridge connecting the simulation and graphics subsystems. A general purpose system library acts as a facade for a variety of low-level system calls and allows the other modules to seamless operate with a threads library, MPI, or on

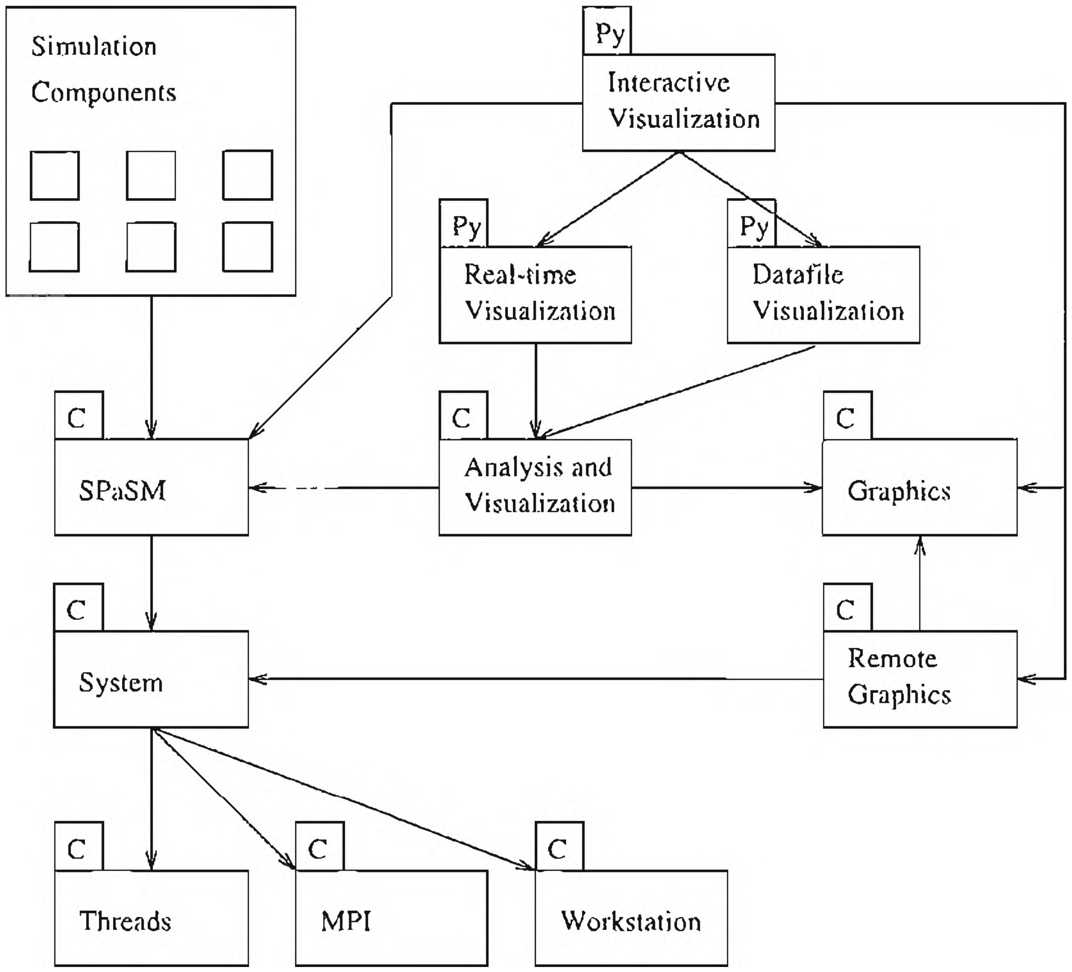


Figure 7.1. SPaSM component architecture

Table 7.1. SPaSM component implementation

Component	ANSI C (lines)	Python (lines)
SPaSM Library	7300	-
System Library	2400	-
Graphics	11000	-
Analysis & Visualization	2700	-
Remote Graphics	550	-
Interactive Visualization	-	2000
Real-time Visualization	-	1000
Datafile Visualization	-	650

a single processor workstation. An interactive data analysis and visualization module is implemented entirely in Python and provides a common interface to two different visualization components one for analyzing running simulations in real time, and one for post-processing data files. Finally, a few special purpose libraries such as a remote graphics module provide additional functionality to allow the graphics subsystem to send images over a socket connection.

7.5.2 Using the System

When a user runs SPaSM, the Python interpreter is started and a number of the core modules are loaded. Afterwards, the user is presented with a command prompt. For example,

```
Python 1.4 (Jan  3 1998)  [GCC 2.7.2.1]
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam

SPaSM ==== Run 41 on guinness ==== Sun Apr 26 16:08:46 1998

Copyright (C) 1992-1997
Regents of the University of California

Loading 'startup.py'
SPaSM [41] >
```

At the prompt, the user can issue commands, execute scripts, and execute any valid Python program. Specific simulations are packaged as Python modules and can be loaded and run interactively. However, common operations are likely to be placed in a script as follows:

```
# Shock wave problem

from Shock import *
nx             = 15
ny             = 15
nz             = 75
shock_velocity = 8.5
temp           = 0.01
width          = 0.3333         # Width is percent of total z length
r0             = 1.0901733      # Lattice spacing
gap            = 0.10           # Gap (% of z length)
cutoff         = 2.0            # Interaction cutoff

ic_shock(nx,ny,nz,shock_velocity,width,gap,temp,r0,cutoff)
init_lj(1,1,cutoff)
```

```
set_boundary_periodic()
set_path("/r0/beazley")
SPaSM_set_output([X,Y,Z,KE,PE])
timesteps(100,10,50,100)
```

When SPaSM is used interactively, the user can perform both simulation and analysis as needed. For example, after running the 100 timesteps in the previous example, the user could load the visualization system and take a look at the data. For example,

```
SPaSM [41] > from vis import *
Colormap set to 'cm_zhou'
Image size set to (400,400)
SPaSM [41] > set_server("slack",1033)
Setting image server to  slack  port  1033
SPaSM [41] > ke = Spheres(KE,0,20)
SPaSM [41] > rotr(90)
SPaSM [41] > rotd(10)
SPaSM [41] > zoom(200)
SPaSM [41] > vz = Profilez(VZ,40,-2,10)
```

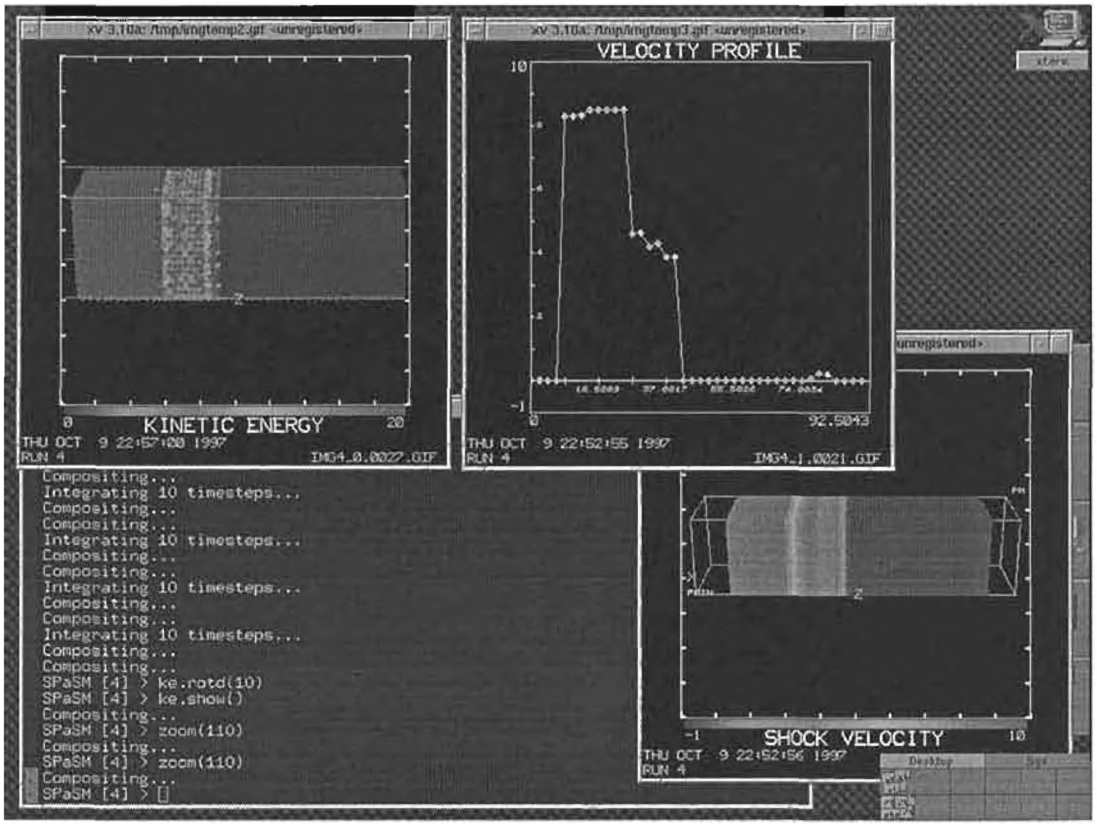
After issuing these commands, images will appear on the user's screen as shown in Figure 7.2. These images represent the current state of the simulation loaded in memory and can be used to watch the progress of the simulation and for diagnostics. In fact, it is even possible to run the simulation and update images simultaneously.

Although the system may appear disorganized and unstructured, this is precisely the desired mode of operation. Rather than having a huge inflexible monolithic application, the code is broken up into small modules that can be used as needed. Furthermore, the Python interface allows users to experiment with the code. Functions can be executed, variables queried, scripts executed, and simulations performed in an interactive and exploratory fashion.

7.5.3 Writing User Code

SPaSM is now controlled through Python, but most users still write code in C. This code includes force functions, boundary conditions, initial conditions and numerical integrators. Even though many of these functions could be written in Python, they are written in C because they are numerically intensive and performance critical.

Problem specific code is maintained separately by each user of the system. Thus, one user may be studying shock waves while another is studying dislocation dynamics. For each simulation, the user simply compiles their problem specific code into a Python



```
Compositing...
Integrating 10 timesteps...
Compositing...
Compositing...
Integrating 10 timesteps...
Compositing...
Compositing...
Integrating 10 timesteps...
Compositing...
Compositing...
Integrating 10 timesteps...
Compositing...
Compositing...
SPaSM [4] > ke.rotd(10)
SPaSM [4] > ke.show()
Compositing...
SPaSM [4] > zoom(110)
Compositing...
SPaSM [4] > zoom(110)
Compositing...
SPaSM [4] > 
```

Figure 7.2. Sample SPaSM session

extension module. This module is then linked against the core SPaSM libraries and used like other modules in the system. In principle, an infinite number of user modules can be created. These modules can be maintained independently and two users can be using SPaSM simultaneously with entirely different problems.

In the previous section, an example involving shock waves was described. This problem consists of approximately 4000 lines of ANSI C code that define all of the physical properties of the problem. To create a Python module, the functions specific to this problem are placed in a SWIG interface file that is converted to Python wrappers during compilation.

Since each problem is defined by a relatively small amount of self-contained C code, the development of modules is considerably easier than working with a large monolithic package. Another important point is that by working with small modules, these modules can be quickly compiled and modified. For example, the entire shock wave problem mentioned above can be recompiled and linked with full optimization in less than a minute on a Sun Ultra-1 workstation. Although the compilation time of modules may seem like a minor point, scientists spend a tremendous amount of time making minor modifications, recompiling their applications, and testing the outcome. For example, a single user of the current SPaSM system was recently observed to have recompiled their simulation module 340 times in a month. Had the compilation and linking process taken 15 minutes, this user would have wasted 85 hours staring at their screen (a startling figure considering that there are only 40 hours in a work week).[1]

7.5.4 Python Programming

One of the most surprising aspects of the system is that significant functionality can be implemented entirely in Python. Since Python is packaged with a large collection of modules and extensions, these can be utilized to create interesting extensions to the molecular dynamics system. A few of these extensions are now described.

7.5.4.1 Web Based Simulation Monitoring

Even with the scripting interface, production simulations may run for tens to hundreds of hours. During this time, it is generally impossible for the user to monitor the progress of a simulation or to view preliminary results (although the visualization system can be set up to generate image files on a periodic basis). To provide this capability, a simple web-server has been implemented entirely in Python. The web-server utilizes the SPaSM visualization module, a variety of Python network modules, and is implemented in fewer than 200 lines of code.

Using the web-server extension, the scientist registers images and files with the server before starting a simulation. A simple numerical integration loop is then written in Python. Inside this loop, a polling operation is performed to see if any users have connected to the code with a web-browser. If so, these requests are serviced before continuing on with the simulation. A simple example is as follows:

```
# Load the shock wave problem
execfile("shock.py")

# Load the visualization module
from vis import *
```

[1]Unfortunately, the developers of sophisticated object-oriented frameworks do not seem to appreciate compilation speed since 30 minute (or even 10 hour) compilation times seem to becoming increasingly common [84].

```
# Create some images
imagesize(450,450)
ke = Spheres(KE,0,20)
ke.rotr(90)
ke.rotd(20)
ke.zoom(160)
ke.title = "Kinetic Energy"

vz = Profilez(VZ,60,-1,10)
vz.title = "Velocity Profile"

shear = PlotCells(SHEAR,-10,10)
shear.title = "Shear Stress"
shear.smooth=1
shear.copyview(ke)

# Create a link to the web-server on my web-page
spasmweb.linkfile("/home/grad/beazley/.public_html/spasm.html")

# Add the images to the web server
ke.web("kinetic.gif")
vz.web("velocity.gif")
shear.web("shear.gif")

# Now run the simulation and periodically poll
for i in range(0,5000):
        timesteps(1,10,50,100)  # Integrate
        spasmweb.poll()         # Check for a connection
```

When this script is executed, the user can point a web-browser directly to the running simulation code. Through the browser interface, the user can examine output files of the simulation as well as generate images. When an image request is made, the visualization module produces a plot and sends it directly to the client. No temporary files are created nor does the server feed previously generated images to the user. Furthermore, no conventional web-server is running on the simulation machine–the user connects to the physics code directly.

Although simple to use and implement, this extension provides a feature not available in other versions of the code. Even more surprising is the fact that it is implemented entirely in the scripting language interface. In other words, the web-browsing feature required no modifications to the underlying C code or other parts of the system. Currently the web-monitoring feature is being used to remotely monitor large-scale simulations running on the ASCI SGI Origin 2000 machines at Los Alamos National Laboratory.

7.5.4.2 Code Browsing

One problem with breaking SPaSM into libraries and modules is the problem of finding functions. For example, a user might want to look at a C function to see what it does or make a modification. Unfortunately, this is difficult if the user does not know, or remember, where the function is defined.

To solve this problem, a simple code browser was implemented entirely in Python. The browser uses Python regular expressions to scan files and directories for function declarations. Thus, using this module, a user can search for files and examine their contents as follows:

```
SPaSM [47] > import browse
SPaSM [47] > browse.find("timesteps")
*****************************
******  ./timesteps.c  ******
*****************************

extern  void     set_boundary_periodic(void);
extern  void     init_lj(double,double,double);
extern  void     integrate_first_velocity(void);
extern  void     integrate_adv_coord(void);
extern  void     integrate_adv_velocity(void);

#define TIMESTEP_TIMER    20

int
timesteps(int nsteps, int energy_n, int output_n, int checkp_n) {
    int             laststep;
    static int      print_log = 0;

    /* Check to make sure it's okay to proceed */

    if (!InitGeom) {
        SPaSM_fprintf(stdout,"No geometry initialized.\n");
        return -1;
--More--(14%)
```

A similar capability is also available for file editing as well. In this case, when a function is found, an editor process (such as emacs or vi) is spawned with the appropriate source file.

7.5.4.3 Distributed Objects

In Chapter 3 distributed object systems such as CORBA and COM were described as alternative approaches to scripting. As it turns out, many scripting languages can also be used in a distributed object framework. For example, the ILU system provides bindings to Python and allows distributed object servers and clients to be implemented entirely in Python [60].

As an experiment, ILU was used to build a distributed object binding to the SPaSM visualization system. This was done by creating a Python wrapper class around the image objects defined in the visualization module. This class was then published on the network using the ILU extension to Python. Once published on the network, remote clients could connect to the visualization system to create and manipulate images.

One such client is a simple Tcl/Tk graphical user interface tool that allows users to interactively drive the visualization system. The client side of the ILU interface was implemented in ANSI C. The functions in this interface were then interfaced to Tcl/TK using SWIG. In Tcl/Tk, a simple GUI was written to allow the user to manipulate images using the mouse in a manner similar to that found in a more conventional visualization package.

Although this is a more complicated example, it illustrates the power of the scripting interface. The ILU network interface is implemented in less than 100 lines of Python and requires no modifications to any other parts of the SPaSM system. Yet, by using such an interface it became possible to drive SPaSM remotely using other languages such as C, C++, Java, Tcl/Tk, and so forth.

7.6 Performance

Prior to adding the scripting interface, SPaSM was a highly tuned application designed to achieve optimal performance on a variety of platforms [64, 15]. In Section 5.8.1, the performance of scripting interpreters was described. In this section, the performance impact of scripting languages on the SPaSM code are described. In addition, the entry of SPaSM in the 1998 Gordon Bell Prize competition is briefly discussed.

7.6.1 Scripting for Control, C for Performance

In SPaSM, scripting languages are primarily used as a control mechanism while computationally intensive operations are written in C. When running a simulation, most CPU is spent calculating interatomic forces–a task that typically requires billions to hundreds of billions of floating point operations. Even less computationally intensive tasks such as numerical integration require hundreds of thousands to millions of floating point operations. The fact that such operations might have been initiated by a slow scripting language interpreter is of little consequence.

To illustrate the inconsequential performance impact of scripting, the outer loop of a simulation was written entirely in Python and compared to an equivalent implementation written in C. The performance impact was then measured for a small simulation as shown in Table 7.2. Even for a small number of atoms, the performance impact is observed to be less than 0.2%.

7.6.2 A Recent Performance Study

SPaSM was recently entered into the 1998 Gordon Bell Prize competition for achieving a price performance of $15/Mflop on a 70 processor DEC Alpha Linux cluster at Los Alamos National Laboratory [102]. This represents a factor 3 improvement in price performance over the 1997 Gordon Bell Prize winner in this category [103].

To achieve this result. the scripted version of SPaSM was used to perform a 60.8 million atom molecular dynamics simulation of shock-induced plasticity in an fcc-crystal structure. This simulation ran on 68 nodes for 2000 timesteps and required approximately 44 hours of simulation time. The simulation also included periodic computation of energies, used the visualization module to create GIF images, and saved 68 Gbytes of check-pointed simulation data to disk. For the entire simulation, SPaSM performed 1.56×10^{15} floating point operations over a wall clock time of 1.58×10^{5} seconds. This corresponds to a sustained throughput of 9.9 Gflops over 44 hours.

Each processor in this system is a 533 Mhz DEC Alpha 21164A with a peak performance of 1.066 Gflops. The entire production simulation (including visualization and I/O) sustains a performance of approximately 146 Mflops whereas the raw force computation runs at 189 Mflops. These performance numbers compare favorably with prior results on a Cray T3D in which an unscripted version of SPaSM achieved a performance of 27-41

Table 7.2. Execution time (seconds) of C versus C with scripting

Atoms per processor	C	C with Python
13950	98.7	98.9
45000	314.1	314.8
180000	1317.1	1319.0

Mflops on a 150 Mhz DEC Alpha [12].

7.7 Results

Scripting languages and SWIG have revolutionized the use SPaSM and made it possible to perform large-scale molecular dynamics simulations on an everyday basis. By improving usability and integrating tasks that were traditionally decoupled, scientists have been able to run simulations and interpret results in a more efficient manner. In fact, it is now common for scientists to perform more simulations in a month than were performed in the first 3 years of the project combined! More importantly, the new system has had a direct impact on scientific results reported in peer-review journals including *Physical Review Letters* and *Science* [110, 111, 58].

On the software side, SPaSM has been transformed from an inflexible monolithic package to a highly modular and flexible component-based system. Along the way, the structure and reliability of the code improved greatly. This has simplified maintenance and allowed the code to be easily expanded in new directions. Finally, the power of scripting languages has even allowed SPaSM to be used in ways not previously imagined.

SWIG played an integral, but unusual, role in the transformation process by greatly simplifying the creation of scripting language modules. SWIG allowed the original application to be easily incorporated into a scripting environment and enabled users and developers to focus on the use and structure of the application, not the gory details of component construction. SWIG also allowed incremental changes and improvements to be easily incorporated into the scripting interface. Even today, SWIG remains a simple mechanism that can be used to extend SPaSM's scripting environment with new capabilities.

CHAPTER 8

USER STUDY

Versions of SWIG have been available for public use since February 1996. Feedback from users has been instrumental in the development of SWIG, and many of its features have been implemented in direct response to user requests. As of May 1998, approximately 360 users subscribed to a SWIG mailing list (swig@cs.utah.edu). To find out more about how SWIG is being used, two user surveys have been conducted. An informal survey, conducted in August 1997, asked mailing list subscribers to describe the applications in which they were using SWIG. A formal survey, conducted over a 7-week period from February to April 1998, asked users a series of questions regarding their use of SWIG and background. This chapter describes the results of these surveys and hopes to provide a picture of how SWIG is being used, who is using it, the benefits it provides, and limitations.

8.1 Survey Methodology

An initial survey asked mailing list subscribers to describe the applications in which they were using SWIG. Approximately 25 responses were received via e-mail, but no statistical data were generated. However, responses to this survey provided some insight into how SWIG is being used. A second survey was conducted over a seven week period to collect statistical data and additional feedback. This survey was conducted through the use of a web-page and CGI script for data tabulation. Respondents were asked a series of questions regarding their use of SWIG as well as their background so that a user-profile could be generated. A full version of the survey can be found in Appendix C. Given the personal nature of many questions and to encourage participation, users were allowed to submit survey responses anonymously although Internet domain names were recorded so that duplicate submissions could be checked and eliminated from the survey if necessary. In addition, most questions were optional–allowing users to skip questions for which they had no opinion (or felt unqualified to offer a valid response). One hundred nineteen responses were received from 114 unique Internet domains. Responses were solicited from the SWIG mailing list and the web-page containing the SWIG mailing list archives. 82% of the respondents subscribed to the mailing list, representing a response rate of approximately 30% for the number of subscribers at the time of the survey.

8.2 User Profile

Since SWIG was originally developed for scientific applications, respondents were asked if they worked on scientific applications. Seventy-two respondents (60%) answered "yes," 46 (39%) answered "no," and one offered no response. For the purposes of further discussion, survey results are divided into the categories of "all users," "scientific users," and "other users."

To get a better idea of who is using SWIG, users were asked questions about their programming experience and background as shown in Table 8.1. Among all users, more than 90% indicated five or more years of programming experience and that number rises to 98% for scientific users. Sixty-six percent of the users classified themselves as software engineers although a majority (57%) do not have a formal degree in computer science. In addition, nearly 70% of the users have, at one time, been a system administrator.

As an additional measure of programming experience, users were also asked about certain types of applications and tools as shown in Table 8.2. The purpose of this table is to find out what kind of tools users might be using in the software development process and application building. From this table, we see that a significant number of users are utilizing makefiles, revision control, configuration management, and other aspects of maintaining complex software packages. There is also a clear distinction between scientific and nonscientific users in a number of categories–especially those associated with distributed object systems and software development on the Windows platform.

Finally, users were asked a few questions about their background with SWIG as shown in Table 8.3. Approximately 60% of respondents have been using SWIG for more than 6 months. Furthermore, nearly all users indicated that they were using scripting languages before using SWIG.

Based on the profile data, the current users of SWIG could probably be categorized as experienced programmers or possibly "early adopters." Most respondents have had significant prior programming experience and nearly all have used scripting languages prior to using SWIG. The fact that a majority of users have been system administrators

Table 8.1. User programming experience and background

Question	All Users (n=119)	Scientific (n=72)	Other (n=46)
How long have you been programming?			
0-5 years	10 (8%)	2 (2%)	8 (17%)
5-10 years	39 (32%)	27 (37%)	12 (26%)
10-15 years	35 (29%)	25 (35%)	10 (21%)
15-20 years	18 (15%)	8 (11%)	10 (21%)
> 20 years	16 (13%)	9 (12%)	6 (13%)
How would you characterize your work?			
Commercial software development	40 (33%)	9 (12%)	31 (67%)
Academic	45 (37%)	40 (55%)	5 (10%)
Government	7 (5%)	5 (6%)	2 (4%)
Industrial research and development	24 (20%)	17 (23%)	6 (13%)
Self employed	2 (1%)	1 (1%)	1 (1%)
Are you a software engineer?			
Yes	79 (66%)	38 (52%)	40 (86%)
No	40 (33%)	34 (47%)	6 (13%)
Do you have a computer science degree?			
Yes	50 (42%)	27 (37%)	23 (50%)
No	68 (57%)	44 (61%)	23 (50%)
Have you ever been a system administrator?			
Yes	83 (69%)	47 (65%)	35 (76%)
No	36 (30%)	25 (34%)	11 (23%)

Table 8.2. User programming experience (applications)

Question	All Users (n=119)	Scientific (n=72)	Other (n=46)
Have you ever written a graphical user interface?			
Yes	103 (86%)	66 (91%)	36 (78%)
No	14 (11%)	5 (6%)	9 (19%)
Have you ever written a network application?			
Yes	79 (66%)	40 (55%)	38 (82%)
No	38 (31%)	30 (41%)	8 (17%)
What other packages/tools do you use?			
Make	107 (89%)	62 (86%)	44 (95%)
Revision control (e.g. RCS)	91 (76%)	52 (72%)	38 (82%)
Configuration tools (e.g. autoconf)	39 (32%)	25 (34%)	13 (28%)
Purify	43 (36%)	24 (33%)	19 (41%)
CORBA	16 (13%)	6 (8%)	10 (21%)
COM	19 (15%)	5 (6%)	14 (30%)
ILU	17 (14%)	9 (12%)	8 (17%)
Visual Basic	23 (19%)	6 (8%)	17 (36%)
Other scripting tools	16 (13%)	10 (13%)	5 (10%)
MATLAB, Mathematica, etc...	49 (41%)	42 (58%)	7 (15%)
Database packages	48 (40%)	21 (29%)	26 (56%)
MPI	10 (8%)	8 (11%)	2 (4%)
Threads	44 (36%)	23 (31%)	20 (43%)
OpenGL	34 (28%)	29 (40%)	5 (10%)
Java	52 (43%)	31 (43%)	21 (45%)

Table 8.3. SWIG experience

Question	All Users (n=119)	Scientific (n=72)	Other (n=46)
How long have you been using SWIG?			
0-6 months	43 (36%)	21 (29%)	21 (45%)
6-12 months	40 (33%)	22 (30%)	18 (39%)
12-18 months	24 (20%)	17 (23%)	7 (15%)
18-24 months	8 (6%)	8 (11%)	0 (0%)
> 24 months	4 (3%)	4 (5%)	0 (0%)
Did you use scripting languages before SWIG?			
Yes	115 (96%)	70 (97%)	44 (95%)
No	4 (3%)	2 (2%)	2 (4%)

also suggests that users are reasonably familiar with the installation, maintenance, and use of various tools and packages.

It is questionable as to whether SWIG is being used by "typical" computational scientists as described in Chapter 2. Generally speaking, scientists are reluctant to adopt unproven software technology and SWIG is no exception. Thus, although 60% of SWIG users work on scientific applications, these users appear to be fairly experienced with respect to programming tools and techniques.

8.3 Languages

Users were asked to indicate which compiled languages as well as scripting languages they were using with SWIG as shown in Table 8.4. For these questions, users were allowed to select all languages that applied. Thus, the data suggest that many users are working with both C and C++ code. Ten percent of users are also working with Fortran even though SWIG currently provides no native support for Fortran. Use of Perl, Python, and Tcl is evenly split among users. The 5% of users using an "other" scripting language are using versions of SWIG that have been extended with additional language modules.

8.4 Using SWIG

To find out how SWIG is being used, users were asked the questions in Table 8.5. Approximately 80% of users are using SWIG to work with C programs containing fewer than 250 functions. This indicates that most users are working with small to moderately sized systems (as a point of reference, the SPaSM code described in Chapter 7 contains approximately 250 C functions). In addition, 50% of users report using SWIG frequently (daily or weekly) and 50% appear to use SWIG only occasionally (monthly or rarely). This split may represent two common ways in which SWIG can be used. One common use is to provide a scripting interface to software that is under development. In this case, the scripting interface is a useful mechanism for debugging, prototyping, and working with the code. Since the interface to such applications would be likely to change frequently, SWIG would be used on a regular basis. A second application of SWIG is the construction of scripting interfaces to existing software packages. For example, a user might build a scripting interface to OpenGL to have an interactive graphics tool. Since OpenGL wouldn't change much at all (if ever), there is little need to run SWIG frequently. Therefore, SWIG might only be used once in awhile to build interfaces to existing packages

Table 8.4. Languages being used with SWIG

Question	All Users (n=119)	Scientific (n=72)	Other (n=46)
What compiled languages do you use with SWIG?			
C	95 (79%)	58 (80%)	36 (78%)
C++	76 (63%)	53 (73%)	22 (47%)
Objective C	2 (1%)	1 (1%)	1 (2%)
Fortran	13 (10%)	12 (16%)	1 (2%)
What scripting languages do you use with SWIG?			
Perl	45 (37%)	22 (30%)	22 (47%)
Python	50 (42%)	35 (48%)	15 (32%)
Tcl	54 (45%)	36 (50%)	17 (36%)
Guile	4 (3%)	3 (4%)	1 (2%)
Other	6 (5%)	5 (6%)	1 (2%)

Table 8.5. SWIG usage

Question	All Users (n=119)	Scientific (n=72)	Other (n=46)
Approximately how large are your interfaces?			
0-49 functions	52 (43%)	32 (44%)	20 (43%)
50-99 functions	24 (20%)	11 (15%)	13 (28%)
100-249 functions	18 (15%)	15 (20%)	3 (6%)
250-499 functions	14 (11%)	9 (12%)	4 (8%)
500-999 functions	7 (5%)	4 (5%)	3 (6%)
More than 1000 functions	3 (2%)	1 (1%)	2 (4%)
What input do you give to SWIG?			
Separate interface files	72 (60%)	41 (56%)	30 (65%)
Header files only	1 (0%)	1 (1%)	0 (0%)
A mix of interface and header files	46 (38%)	30 (41%)	16 (35%)
How do you run SWIG?			
From the command line	22 (18%)	11 (15%)	11 (23%)
From a makefile	89 (74%)	57 (79%)	31 (67%)
Development environment	7 (5%)	4 (5%)	3 (6%)
How often do you use SWIG?			
Daily	24 (20%)	12 (16%)	11 (23%)
Weekly	36 (30%)	26 (36%)	10 (21%)
Monthly	27 (22%)	17 (23%)	10 (21%)
Rarely	32 (26%)	17 (23%)	15 (32%)

as necessary.

Table 8.6 shows the usage of various SWIG features. Although the use of various features is not particularly relevant for the purposes of this discussion, the data provides some information about the parts of SWIG that are actually being used.

Finally, users were asked about the process of compiling SWIG generated modules as shown in Table 8.7. An increasingly common problem with many modern software systems is that of long compilation times. Sixty-five percent of users report that it takes less than a minute to run SWIG and compile the wrapper code into a module. However, a small percentage of users report times longer than 10 minutes and some of these users report cases in which SWIG has produced wrapper modules that were too large to be compiled by the C or C++ compiler. In addition, 43% of users reported that they have had to modify the output of SWIG at some time. This statistic will be discussed later in this chapter.

8.5 Evaluation

The statistical survey also included an evaluation section. In this section, users were asked to evaluate various statements and assign points on a scale of 1 (disagree) to 5 (agree). The results of the evaluation section are shown in Table 8.8. Although it is difficult to read much into the evaluation results since the survey is biased (users who hate SWIG were unlikely to spend time filling out a survey), the evaluation section does indicate relative weaknesses and strengths of the current implementation. In particular, users generally feel that SWIG is easy to use and install. However, the documentation generation system and the claim that SWIG requires no modification to underlying code receive relatively low marks. On the positive side, users strongly agree that SWIG and scripting have had a positive impact on their programming projects and have given this statement the highest average score.

8.6 Application Areas

To find out what SWIG is being used for, users were asked about a number of general software development activities shown in Table 8.9. In addition, users were asked for a short description of their application area. Table 8.10 provides a short summary of the application areas received in the survey.

A majority of SWIG users are involved in research and development projects. Furthermore, there seem to be three primary areas of applicability. First there are applications

Table 8.6. SWIG feature usage

Question	All Users (n=119)	Scientific (n=72)	Other (n=46)
What SWIG features do you regularly use?			
File inclusion	80 (67%)	54 (75%)	26 (56%)
Typemaps	76 (63%)	42 (58%)	33 (71%)
Wrapper classes	56 (47%)	34 (47%)	22 (47%)
Documentation generation	46 (38%)	30 (41%)	16 (34%)
Class extension	43 (36%)	28 (38%)	15 (32%)
Renaming	43 (36%)	29 (40%)	14 (30%)
Runtime libraries	30 (25%)	18 (25%)	12 (26%)
Exception handling	20 (16%)	11 (15%)	9 (19%)
What SWIG library files do you regularly use?			
Typemaps	60 (50%)	37 (51%)	22 (47%)
Pointers	37 (31%)	27 (37%)	10 (21%)
Exception	14 (11%)	8 (11%)	6 (13%)
Constraint	4 (3%)	4 (5%)	0 (0%)
What SWIG documentation formats do you use?			
HTML	48 (40%)	29 (40%)	19 (41%)
ASCII	22 (18%)	10 (13%)	12 (26%)
LaTeX	10 (8%)	9 (12%)	1 (2%)
None	38 (31%)	23 (31%)	14 (30%)
Have you used SWIG with more than one scripting language?			
Yes	30 (25%)	21 (29%)	8 (17%)
No	88 (73%)	50 (69%)	38 (82%)
Have you ever created a new language module or modified the SWIG source code?			
Yes	18 (15%)	8 (11%)	10 (21%)
No	99 (83%)	63 (87%)	35 (76%)

Table 8.7. Compilation of SWIG generated extensions

Question	All Users (n=119)	Scientific (n=72)	Other (n=46)
Approximately how long does it take to build a SWIG extension on your machine?			
0-30 seconds	53 (44%)	32 (44%)	20 (43%)
30-60 seconds	26 (21%)	19 (26%)	7 (15%)
1-2 minutes	22 (18%)	11 (15%)	11 (23%)
2-5 minutes	9 (7%)	5 (6%)	4 (8%)
5-10 minutes	1 (0%)	1 (1%)	0 (0%)
More than 10 minutes	5 (4%)	1 (1%)	4 (8%)
Have you ever generated an extension module that was too large to be compiled?			
Yes	6 (5%)	2 (2%)	4 (8%)
No	111 (93%)	68 (94%)	42 (91%)
How do you typically link modules?			
Shared libraries and dynamic loading	92 (77%)	56 (77%)	35 (76%)
Static linking	27 (22%)	16 (22%)	11 (23%)
Have you ever modified the wrapper code generated by SWIG?			
Yes	52 (43%)	29 (40%)	22 (47%)
No	67 (56%)	43 (59%)	24 (52%)

Table 8.8. SWIG evaluation

Question	All Users (n=119)	Scientific (n=72)	Other (n=46)
SWIG is easy to install	4.42 (σ=0.84)	4.51 (σ=0.77)	4.30 (σ=0.93)
It was easy to build your first SWIG example	3.97 (σ=1.09)	4.08 (σ=0.97)	3.83 (σ=1.20)
In practice, SWIG is easy to use	4.11 (σ=0.90)	4.19 (σ=0.84)	4.00 (σ=0.96)
The scripting interfaces created by SWIG are easy to use	4.30 (σ=0.73)	4.43 (σ=0.70)	4.11 (σ=0.73)
SWIG generated modules can be quickly compiled	4.04 (σ=1.02)	4.09 (σ=0.92)	3.96 (σ=1.16)
SWIG requires no modifications to the underlying C/C++ code	3.72 (σ=1.06)	3.74 (σ=0.97)	3.74 (σ=1.17)
Parsing ANSI C/C++ declarations makes SWIG easier to use	4.26 (σ=1.04)	4.36 (σ=0.97)	4.11 (σ=1.13)
SWIG allows you to build interfaces without worrying about the details	4.28 (σ=0.89)	4.32 (σ=0.89)	4.22 (σ=0.89)
The documentation files created by SWIG are useful	3.55 (σ=1.07)	3.59 (σ=1.04)	3.51 (σ=1.11)
SWIG/Scripting has had a positive impact on your projects	4.63 (σ=0.64)	4.67 (σ=0.58)	4.60 (σ=0.68)

Table 8.9. General uses of SWIG

Question	All Users (n=119)	Scientific (n=72)	Other (n=46)
Personal use	63 (52%)	42 (58%)	21 (45%)
In-house application development	53 (44%)	31 (43%)	22 (47%)
Software testing and debugging	27 (22%)	15 (20%)	11 (23%)
Research and development projects	62 (52%)	49 (68%)	13 (28%)
Rapid prototyping	42 (35%)	31 (43%)	11 (23%)
Commercial software development	31 (26%)	14 (19%)	17 (36%)
Other	2 (1%)	1 (2%)	1 (2%)

Table 8.10. SWIG application areas

Animation	Astrophysics
Automotive R&D	CAD tools
CASE tools	COM
CORBA	Chemical information systems
Climate modeling	Computational chemistry
Database	Defibrillation modeling
Document management	Drawing
Economics	Education
Electronic Design Automation	Electronic commerce
Financial	Fortran
Games	Groupware
Hardware control/monitoring	Image processing
Integrated Development Environments	Lotus Notes
Materials modeling	Medical imaging
Meteorological imaging	Microprocessor design
Military visualization	Molecular dynamics
Natural language processing	Network management
Neural nets	Oil exploration
Palm Pilot	Polarization microscopy
Protein sequence analysis	Raytracing
Realtime automation	Robotics
Software testing	Spectrographic analysis
Speech recognition	Testing of telecom software
Virtual reality	Vision
Visual simulation	Weather forecasting
X-ray astrophysics analysis	

of an inherently exploratory nature. In this case, SWIG is being used to add a scripting interface to improve the flexibility and usability of these applications. A second area is in the area of systems integration and the use of scripting languages as a means for integrating existing software components into a common framework. Finally, a number of users indicated that they are using SWIG for testing and diagnostic applications. In this case, scripting languages are added to systems for the purposes of in-house testing and debugging but are not delivered with the final product.

8.7 Benefits of Using SWIG

Unfortunately, it is difficult to quantitatively measure the success of SWIG using traditional software engineering metrics. Given that the primary users of SWIG are scientists and that most applications are of a research and experimental variety, users are unlikely to record quantitative information about the development process. Furthermore, SWIG is largely concerned with improving the usability of applications—an important task, but one that is not easily measured. Although the user survey provides some insight into how SWIG is being used and who is using it, it provides little insight into why a user might use SWIG.

This section provides antedotal evidence about why users are using SWIG and the advantages that it offers. This information was collected from the user surveys, messages on the SWIG mailing list, and through personal communications with users. The following testimonials should be viewed with a degree of caution as there is no way to measure if the opinions expressed are representative of everyone who has tried to use SWIG (especially since people who have tried and failed are unlikely to provide any feedback).

To protect the identity of respondents, each respondent has been assigned two letter code. A mapping of codes to respondent names exists, but is not included in the dissertation.

8.7.1 Ease of Use

A number of users are using SWIG because it simplifies the process of building scripting language interfaces. The following quotes are typical.

DM writes,

> SWIG has helped us minimize the hassle of writing manual wrappers. Since SWIG has proven to be rather easy to use, I find I can carry out the types

of wrapping activities which would otherwise have been the responsibility of a computer scientist.

MM writes,

I really love the fact that the learning curve is short and flat. I don't need to use this facility very often, but when I absolutely have to link in external C routines, or when I have to get the last drop of speed from something previously written in Perl, I can count on SWIG to be there. When I do use SWIG, I don't need to spend hours learning and debugging the system. This is the closest thing to cut-and-paste I've ever seen in inter-language library creation.

AA writes,

We are a research group that develops medical imaging software. We are interested in image processing/visualization research, and for our software, we want a simple and portable tool to generate user interfaces. For this, we currently use Tcl/Tk, and SWIG which provides a nice way to connect our software (in C++) to the tcl-scripts.

ME writes,

Very quick first results.

JS writes,

Easy to use, no need to worry about language internals. It is a boon for application developers, like me.

HR writes,

I like the fact that it automates much of the tedious work in building an interface C/C++ functions. This makes development easier.

AB writes,

SWIG is by far the easiest way I have found to generate scripting interfaces for scientific software. SWIG makes it practical to use a single, powerful scripting language for all projects rather than writing a custom interface to each. It also encourages a consistent and modular form for a program, and makes it easier to add contributions to my programs from other users/programmers.

HH writes,

I like the ease with which scripting languages can be extended. The fact that we could so easily interface with Lotus Notes on NT using Python was just amazing.

JH writes,

SWIG makes linking my application to a scripting language easy enough that it is worthwhile.

JK writes,

> I came, I saw, I wrapped. And it ran. Woo hoo!

8.7.2 Productivity

By automating the generation of wrappers, SWIG allows developers to focus on the problems at hand. The following quotes address the improved productivity of using SWIG.

MR writes,

> Without SWIG it would have taken much, much longer for our group to use Python as an extension language. Our whole application was written mostly in C++. We wanted to look into using Python for portions of it in order to make it easier to extend. Python has a fairly nice interface for doing this sort of thing. The trouble was that we needed to make hundreds of C++ classes available to Python. This would have taken a very long time (to write the wrapper code). SWIG allowed us to spend a minimal amount of time with the wrapper code and most of our time moving stuff to Python (which was the big point in the first place).

RD writes,

> SWIG is a huge time-saver. I have approximately 30,000 lines of C and Python code that have been generated by SWIG that I didn't have to write by hand, don't have to fix syntax and fumble-finger errors in, and don't have to aggressively test.

HS writes,

> SWIG helps us in taking away part of the error-prone task of making the C routines accessible from Python and has considerably improved our efficiency.

AC writes,

> I like all the time I have saved by not writing the interfaces myself.

BH writes,

> SWIG saves a lot of my energy in interfacing with many free C/C++ libraries in my project.

AD writes,

> SWIG handles the gory details and allows me to concentrate on the important things.

GM writes,

> Using SWIG is really fun, because it saves you from a lot of mechanical work and it takes care of all the details you don't want to bother with letting you concentrate on the real problem.

RB writes,

> Thanks again for SWIG... It's fun and allows great productivity while avoiding much tedium.

AD writes,

> [I like] the ability to write extensions basically without having to think too hard about what I'm doing.

PD writes,

> SWIG allows me to get on with scripting and writing C++ code without having to worry about the (usually considerable) issues involved in extending the scripting language with my custom components.

8.7.3 Software Development

The use of scripting and SWIG can have a dramatic impact on the software development process by encouraging modularity and providing a powerful debugging and diagnostic capability. The following quotes address some of the software development benefits of using SWIG.

JW writes,

> [SWIG has impacted my software development process as follows.]
>
> - Increased productivity. I no longer do edit, compile, link, and debug but I build a SWIG interface and compile and then use an interpreter. This change of work styles lets me be much more productive when I am doing exploratory programming. If the code that uses the interpreter is too slow, I rewrite a very small portion.
> - More bugs are found. Since I can quickly assemble new programs in different ways, I find bugs sooner. At [company], I found a critical bug that had existed in the software for over 5 years. (The bug was that if a single process opened the same file twice for reading, the file was corrupted.)
> - I was a happier programmer. I don't have to deal with lots of the low level data structures when I am prototyping. If I need a list of widgets, just wrap my widget with SWIG and use Python's list.

PD writes,

> Using a scripting language as glue between C++ components is a powerful paradigm for combining flexibility with robustness and efficiency. SWIG enables this model by providing a solid bridge between the C++ component and the scripting language.

AF writes,

> The ability to "follow" the development of the core application without constantly rebuilding the interface is very effective. The developments of the kernel and its interface are mutually protected to a large degree.

CW writes,

> SWIG allows us to recycle a lot of ugly old C code and put it into a reasonable module structure and snazzy new user interfaces.

KL writes,

> It allows us to leverage the advantages of the scripting language, especially when so many other scripts are already being written to glue programs together and some of our other tools have their own scripting language interface. Using SWIG will allow us to properly integrate each of the parts directly into the language instead of a collection of system() calls.

KR writes,

> Before I had to use C++ for my "rapid" prototyping. Now I can script it!

AF writes,

> My code is cleaner and more compact which makes it easier to read and understand. SWIG also encourages modularize code–allowing one to test/debug modules independently. This makes connecting everything together a breeze.

JM writes,

> The very idea of scripting programming on the one hand and systems programming on the other is quite nice, the most important feature of SWIG is to make this approach practical on a day-by-day basis.

AG writes,

> On the whole, SWIG is my most important development tool after gcc!

8.7.4 Usability

Finally, by using SWIG to build scripted applications, those applications can become more flexible and usable.

MW writes,

> SWIG has enabled our customers to interact with our toolkits in fundamentally new ways.

BT writes,

> We are using Tcl scripts as the data files driving our simulations. Once the data is defined using a program (which is pretty cool in itself), we can actually run

> the simulation from script commands. In our experimental environment, that saves rewriting a lot of "main" programs that exercise the same basic objects. This isn't exactly computational steering, but it does give our engineers a lot of flexibility.

MB writes,

> All (well most) of my C++ code (MC simulations of proteins and sequence analysis) is now driven by a Python interface thanks to SWIG. Once I have decided on an interface, the process of building it is usually trivial.

LB writes,

> SWIG is an integral part of a user environment I am creating for a Molecular Dynamics company. They have FORTRAN modules that require a steering language (Python) to enable flexible computational research.

YZ writes,

> SWIG plays a critical role to automate the generation of Perl client interfaces from the OMG IDLs for a CORBA ORB. The Perl client interface is essential in script driven testing.

LS writes,

> I am enjoying rapidly developing complex projects using OO and Python, but coming from a numerical background, I like that I can get fast number crunching performance when I need it from C modules wrapped up by SWIG.

8.8 Limitations

Even though SWIG is being successfully used in a wide variety of applications, it still has a number of limitations. This section describes some of these limitations and workarounds.

8.8.1 Survey Results

The user survey asked respondents to pick one area for future SWIG improvement from a list of possibilities shown in Table 8.11. Respondents could also provide written comments to elaborate on SWIG limitations. The most significant limitations of SWIG appear to fall into a number of categories. First, there are problems handling certain datatypes such as arrays, pointers to functions, and so forth. Second, there are parsing difficulties because SWIG is not a full C/C++ compiler. Additionally, there are semantic difficulties due to differences between what is possible in C/C++ and what SWIG supports. Finally, there are conceptual difficulties with using certain parts of the SWIG compiler. Several of the more common limitations are now described.

Table 8.11. Areas in which SWIG could be improved

Category	All Users (n=119)	Scientific (n=72)	Other (n=46)
Better support for arrays	26 (21%)	18 (25%)	8 (17%)
Support for overloaded functions	26 (21%)	17 (23%)	9 (19%)
Better C/C++ parsing	16 (13%)	9 (12%)	7 (15%)
Support for Java	14 (11%)	6 (8%)	7 (15%)
Optimized output	11 (9%)	7 (9%)	4 (8%)
An extension mechanism	8 (6%)	2 (2%)	6 (13%)
Support for Fortran	7 (5%)	7 (9%)	0 (0%)
More SWIG library files	4 (3%)	3 (4%)	1 (2%)

8.8.2 Array Handling

By default, SWIG treats all arrays as simple pointers. Since there is a close relationship between arrays and pointers in C, this is a generally effective management technique. Where many difficulties arise is in the interface between arrays and lists in a scripting language and arrays in C. For example, lists of objects can be easily defined and manipulated in Python as follows:

```
Python 1.5 (#1, Jan  1 1998, 11:26:26)  [GCC 2.7.2.1] on linux2
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> a = [1,2,3,4,5,6]
>>> print a[3]
4
>>> print a[0:3]
[1, 2, 3]
>>> ... etc ...
```

Thus, a natural inclination of users is to use a Python list as input to a C function expecting an array. Unfortunately this is not possible because C arrays and Python lists are represented in an entirely different manner and are not interchangeable. Because of this, users often have to manufacture a C array, fill it with values, and pass it to a C function as follows:

```
a = ptrcreate("double",0.0,4)       # Create a double[4]
value = [1,2,3,4]
for i in range(0,4):
        ptrset(a,value[i],i)        # a[i] = value[i]
foo(a)                              # Call a C function
ptrfree(a)                          # Destroy the array
```

Since this can be somewhat awkward, many users prefer to write a typemap for converting scripting arrays to C. Although the typemap approach solves some of the problems using scripting language arrays, additional problems arise especially when functions expect size arguments. For example, some C functions operating on array data are structured as follows:

```
void foo(double *array, int size);
```

Although a typemap rule can be given for the first argument, there is no way for SWIG to expand a single object (such as a Python list) into a pair of function argument values. Thus, the function has to be used as follows:

```
# Call foo, but explicitly specify the array length
foo([1,2,3,4],4)

# An alternative approach
a = [10,11,12,13,14,15,16]
foo(a,len(a))

# An error, no length specified
foo([1,2,3])
```

To automatically fill in size parameters, users can recast functions using a containers and helper function as described in Chapter 5. Unfortunately, doing so can require a substantial amount of extra work if many functions need to be transformed in this manner.

Finally, there seems to be no one representation of arrays that will be sufficient for use with all applications. In scientific applications, users often work with huge amounts of data and arrays involving millions of data points. In these cases, it would be impractical to convert this data to and from a scripting representation in the same manner as might be done for small arrays. Not only would this conversion process be computationally expensive, it would also incur a serious memory penalty since scripting languages require more memory to represent basic objects than C (for example, a double precision floating point number in C requires 8 bytes of storage compared to 16 bytes for Python).

Most of the problems with arrays are due to the huge variety of ways in which arrays can be defined, used, and represented in both C and the target scripting language. It appears to be impossible to devise one single approach that will serve all situations. Therefore, the SWIG approach is to minimally represent arrays as pointers (which is always possible) and to allow the user the ability to customize SWIG as appropriate for an application. Unfortunately, this minimalistic approach leaves array processing largely up to the user (where confusion can result).

8.8.3 Overloaded Functions

SWIG currently provides no support for overloaded functions, but such functions appear frequently in C++ applications. For example,

```
void foo(double);
void foo(int);
void foo(char *);
```

To access all of these functions from scripting, it is necessary to rename them with unique names as follows:

```
// SWIG interface to overloaded functions
%name(foo_double) void foo(double);
%name(foo_int) void foo(int);
%name(foo_char) void foo(char *);
```

Even though this approach works, it makes SWIG awkward to use with some C++ programs. Support for overloading also presents a number of challenges. The most difficult challenge is that of developing a type disambiguation scheme that selects and executes the appropriate C++ function based on the arguments passed to a wrapper function. In C++, type-signatures are used to resolve overloaded functions, but most scripting languages utilize types in an entirely different manner. In fact, languages such as Tcl might represent all data as strings. Therefore, the Tcl function call "foo 4" could legally invoke all three versions of the underlying C++ function since "4" is simultaneously a string, a float, and integer in the Tcl. As a result, overloading can be difficult to support in full generality. However, a number of specialized extension building tools such as the interface builder for VTK have shown that overloading can be supported for certain applications [89].

8.8.4 Better C++ Support

Although SWIG supports a simple subset of C++ and can build interfaces to C++ programs, it has trouble with more advanced C++ features. Templates are only supported in a limited manner, operator overloading is entirely unsupported as are C++ namespaces. As a result, the process of building SWIG interfaces to C++ programs can be considerably more time consuming than for C programs.

The survey asked C++ users to indicate which C++ features they were using. These results are shown in Table 8.12. Based on these results, it is clear that improving SWIG's C++ support will be beneficial. However, doing so is no easy task. Not only will better C++ parsing be required, many C++ features are not easily integrated in a scripting environment. For example, templates make no sense to a scripting interface, but a scripting interface to a specific template instantiation might prove useful. The variations between C++ compilers also complicate matters (however this situation appears to be improving now that an ANSI C++ standard has been approved).

8.8.5 Code Optimization

Although SWIG is relatively easy to use, it can sometimes produce a substantial amount of wrapper code. In the case of the SPaSM code described in Chapter 7, approximately 30000 lines of wrapper code are generated to build the Python interface. For large applications, the amount of wrapper code can be staggering (hundreds of thousands of lines of code). Compiling this code can be problematic and can even push the limits of existing compilers. A few users reported compile times of greater than 10 minutes when creating a SWIG module. A few other users reported that the SWIG generated wrappers were too large to be compiled (and resulted in an internal compiler error).

Given that most users are building small to moderately sized interfaces, the size of the wrapper code does not yet appear to be a widespread problem. However, there is significant interest in improving SWIG to make it produce less wrapper code. If the amount of wrapper code can be reduced, it will decrease compile times and increase SWIG's applicability to larger applications.

Table 8.12. C++ features being used by SWIG C++ users

Category	All Users (n=76)	Scientific (n=53)	Other (n=22)
Templates	58 (76%)	40 (75%)	18 (82%)
Namespaces	14 (18%)	7 (13%)	7 (32%)
Exceptions	32 (42%)	19 (36%)	13 (59%)
Operator overloading	53 (70%)	38 (72%)	15 (68%)
Standard template library (STL)	40 (53%)	29 (38%)	11 (50%)
"Smart" pointers	14 (18%)	10 (19%)	4 (18%)
Expression templates	4 (5%)	4 (8%)	0 (0%)

8.8.6 Is SWIG Automatic?

In the survey, 43% of users indicated that they have modified the wrapper code generated by SWIG. This is a rather surprising result considering that one of SWIG's goals is to completely automate the interface construction process. Based on mailing list discussions, it appears that users often attempt to customize SWIG's behavior by hand editing the resulting wrapper code. However, these users are often surprised to find out that the exact same modifications can be implemented using typemaps or some other SWIG customization option. Therefore, the high number of users making modifications may be due to confusion and conceptual difficulties regarding SWIG's current customization options and internal operation (it may also just be an issue of documentation).

8.8.7 Conceptual Barriers

Certain aspects of SWIG, especially those pertaining to customization, have created a considerable amount of confusion among certain SWIG users. One such area is the implementation and use of typemaps. Using typemaps, a user can customize SWIG's processing in any almost any imaginable manner. However, doing so requires an intimate knowledge of C, the original application, SWIG, and the target scripting language. To further complicate matters, errors in typemap definitions can result in bizarre errors and scripting interfaces that are impossible to use. Thus, user sentiment seems to range from "I hate typemaps, but I like their functionality" to "typemaps are wonderful!"

The other conceptual problem is related to the use of SWIG, scripting, and C in general. Many users have not used C/C++ code in this manner before. As a result, there is a learning process involving in figuring out how to compile and link modules, how the scripting interface works, and how all of the pieces of the system interact with each other. This is not necessarily a limitation of SWIG, but indicative of the fact that building scripted applications is quite different than building simple stand-alone programs.

8.9 Summary

The survey indicates that a majority of SWIG users are fairly experienced and sophisticated programmers. In fact, it is quite likely that many of the users could be considered to be "early adopters" and not necessarily representative of the scientific computing community as a whole. The survey also indicates that SWIG is generally easy to use and that it has had a positive impact on productivity, software development, and the usability of applications. However, the survey also points out a number of limitations in SWIG's implementation and design. Although these limitations do not appear to be a serious impediment to using SWIG (since workarounds are available), they offer many opportunities for future improvements and development.

CHAPTER 9

RESULTS AND CONCLUSIONS

9.1 Evaluation of SWIG

The case-study and user survey provide strong evidence that SWIG is being successfully used in a variety of applications. Although it is difficult to quantify the reasons why a scientist might use SWIG over other tools and techniques, the following success criteria may hold much of the answer.

Ease of use. Traditionally, the process of creating scripting interfaces has required the development of wrapper code or the use of quirky extension building tools (many of which use special interface definition formats or are limited in capabilities). SWIG, on the other hand, can quickly build scripting interfaces to existing C/C++ programs with very little work. As a result, users are often able to utilize the power of scripting languages almost immediately. This is perhaps best said by one of SWIG's users who writes, "SWIG really helped me get the system off the ground in the shortest amount of time. I never would have believed how easy it was until I wrapped Sun's rpc.cmsd daemon (at least 50 thousand lines of C) with about 20 lines of interface code. Mind-blowing."

Applicability to real software. To be useful, tools need to work with real software packages. Furthermore, they need to be highly adaptable in order to accommodate different programming styles and software designs. In the case-study, SWIG was effectively used with a high-performance application consisting of approximately 25000 lines of code and developed for massively parallel supercomputing systems. In the survey, users indicated that they were using SWIG with a wide variety of scientific packages and commercial systems. Furthermore, several users indicated that they were using SWIG to develop commercial software products. Finally, a number of applications where SWIG has been utilized are starting to appear in the literature [19, 107, 73, 91].

Productivity. SWIG improves productivity by eliminating the need to write scripting extensions by hand and allowing developers to focus on the problem at hand. In fact, a number of survey respondents indicated that SWIG was a tremendous productivity and time-saving tool. One user even wrote, "Without SWIG, it would be almost impossible for me to keep up with my projects."

Performance. Performance is often a deciding factor in the choice to use various tools in the scientific computing community. In the case of SWIG, it has been shown that the use of scripting languages can have a minimal impact on the performance of compiled applications. In the case-study, the performance of the SPaSM code was minimally affected by the addition of a scripting interface. In fact, SPaSM was even recently entered in the 1998 Gordon Bell prize competition for sustaining 10 Gflops performance on Linux cluster [102]. In addition, no survey respondents reported that the use of SWIG and scripting languages had a serious performance impact on their projects.

9.2 The Impact of Scripting Environments

Scripting languages have a huge impact on improving the usability of scientific software because they provide an interpreted high-level environment that simplifies the control and specification of complex problems. This environment allows scientists to use applications in an interactive and exploratory manner. Furthermore, the ability of scripting languages to manage and combine software components allows different packages and tools to be integrated in a shared environment. This integration streamlines the problem-solving process and makes scientists more productive.

In Chapter 7, the dramatic changes and improvements to the SPaSM molecular dynamics code were described. Before the addition of scripting languages, this application was extremely difficult to use and had only been utilized in a small handful of test simulations. Scripting languages made this application usable and enabled scientists to explore large-scale molecular dynamics problems on a daily basis. Today, the SPaSM code is in almost constant use. Furthermore, simulations performed with SPaSM have directly led to a number of results published in peer-review scientific journals. Without SPaSM's scripting language environment and its exploratory capabilities, it is unlikely that these results would have been obtained.

Scripting languages had a tremendous impact on the SPaSM code and the user survey in Chapter 8 suggests that scripting has had a large impact on other applications. In particular, scripting improves the way in which applications are controlled, provides a mechanism for gluing different software components together, and simplifies application development.

9.3 The Role of SWIG

Although scriptable applications can be built by hand, SWIG greatly simplifies the construction of such applications and makes the use of scripting languages practical on a daily basis. In fact, SWIG allows scripting languages to be used in situations where they might otherwise have not been considered. This is possible because SWIG almost completely automates the process integrating scripting languages with compiled code. As a result, users can exploit scripting languages while concentrating their efforts on the problem at hand (not the nasty coding issues associated with creating scripting language extensions).

9.4 Scientific Software Development

Finally, SWIG and scripting languages have a dramatic impact on the development and organization of scientific software. First, SWIG makes it trivial to add a very powerful user interface to most programs as shown in Figure 9.1. In fact, even for simple programs, SWIG allows a scripting interface to be built with less effort than hacking an interface together with input files or command line options. As a result, SWIG can be used effectively with programs large and small as well as with programs at all stages of their development.

The second major area of improvement is in the structure and reliability of scientific software. Since SWIG automatically encapsulates applications in a highly flexible environment, developers can focus their attention to the overall structure and use of an application. When working with existing applications, those applications may undergo an evolutionary process of improvement. In fact, scientists may even implement exceptions, assertions, and other aspects of more advanced software packages. Furthermore, when SWIG is used to develop new applications, scientists can often avoid the pitfalls associated with traditional scientific software development.

Finally, scripting languages encourage the development of modules and software components. Rather than creating huge monolithic applications, packages can be broken up into independent modules. This simplifies application development by dividing appli-

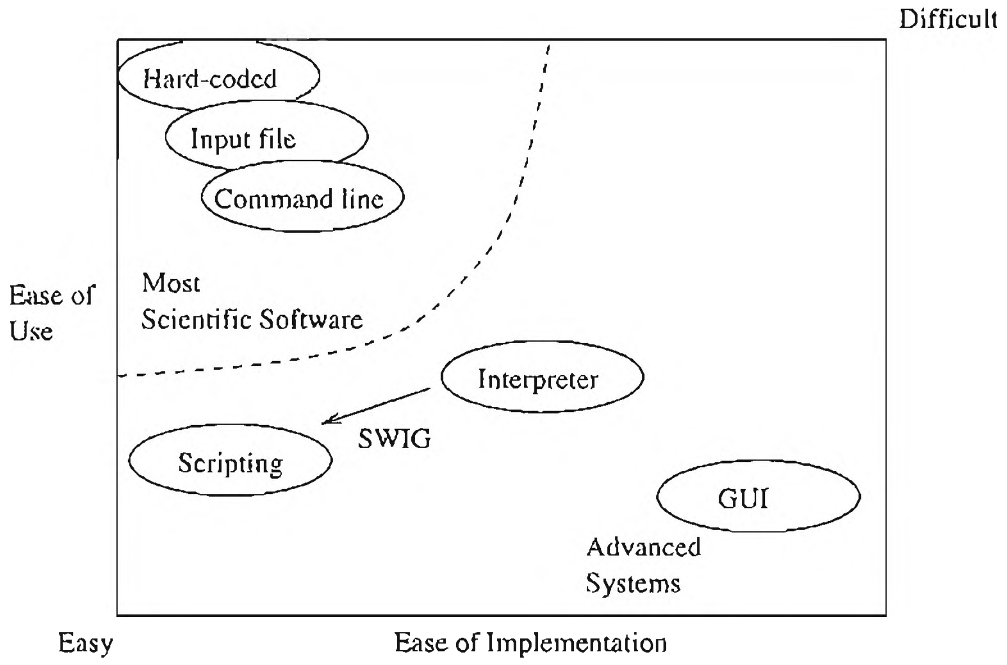


Figure 9.1. User interface ease of use versus implementation difficulty with SWIG

cations into smaller units of functionality that are easier to write and maintain. Furthermore, the component approach also allows different packages to be combined and utilized in a manner not previously possible. For example, the combination of simulation and visualization modules in the SPaSM code allowed scientists to explore data in a substantially more flexible and efficient manner than previously possible.

9.5 Future Challenges

SWIG has proven to be highly effective at building scriptable software. However, there are many areas for future improvement.

Improved Language Support. Given the use of Fortran in the scientific community, adding Fortran support to SWIG would likely be a boon to scientific software developers looking to breathe new life into legacy systems (in fact, some users are using SWIG with Fortran codes even though no native Fortran support currently exists). There are also many opportunities to improve SWIG's support for C++ systems. Unfortunately, many aspects of C++ are particularly difficult to integrate into a scripting environment. However, with improvements to the SWIG parser and code generator, it may be possible to greatly improve C++ in the future.

Component-based Scientific Software. Scripting languages and SWIG allow scientists to create component-based systems. However, the use of software component architectures represents an entirely different methodology of constructing scientific programs. As a result, there are a number of open questions. For example, how do scientists guarantee reproducibility of results in an environment of constantly changing software modules? Likewise, what are the scaling properties of component-based systems? Would the informality and flexibility of SWIG break down as the number of software components increases?

User Studies. SWIG and scripting languages clearly affect the way in which scientific software is developed and used. However, it has proven to be extraordinarily difficult to quantify and measure the overall impact of these techniques. Therefore, there is a considerable need for more case-studies and more detailed user-studies. Unfortunately, this is a difficult task since scientific projects rarely lend themselves to the same analysis techniques that might be used to measure success of a large software engineering effort.

Education. SWIG and scripting languages allow scientific software to be constructed and used in an entirely different manner than that traditionally found in most scientific programs. As a result, there is a considerable need for education among the scientific community. Although the creation of scriptable applications is rarely difficult, it requires a different set of tools and mindset for thinking about the nature of scientific programs.

9.6 Conclusion

Scripting languages offer a flexible environment that can be used to manage complexity and greatly improve the usability of scientific software. SWIG enables scientists to effectively use scripting languages by simplifying the integration of existing code written in compiled languages. This makes it practical for scientists to utilize scripting languages with a wide range of scientific projects on an everyday basis. This, in turn, results in both better scientific software and an a greatly improved environment for solving scientific problems.

APPENDIX A

SCRIPTING LANGUAGE EXTENSIONS

This section shows three scripting-language extension modules for the following C declarations

```
/* A C function */
int foo(int a, double b, char *c);

/* A global variable */
double A;

/* A constant */
#define PI   3.141592654
```

The purpose of this section is to show what scripting languages modules look like and to give a better idea of the code produced by SWIG.

A.1 A Perl Extension Module

```
/*
 * A Perl5 extension module */
 */

#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

/* Application specific headers */
#include "example.h"

/* Wrapper function for foo(int a, double b, char *c) */

XS(wrap_foo) {
    int  result;
    int  arg0;
    double  arg1;
    char * arg2;
    dXSARGS ;
    if (items != 3)
        croak("Usage: foo(a,b,c);");
    arg0 = (int )SvIV(ST(0));
    arg1 = (double ) SvNV(ST(1));
    arg2 = (char *) SvPV(ST(2),na);
    result = (int )foo(arg0,arg1,arg2);
    ST(0) = sv_newmortal();
    sv_setiv(ST(0),(IV) result);
    XSRETURN(1);
}

/* Wrapper functions for variable linking */
static int wrap_set_A(SV* sv, MAGIC *mg) {
    A = (double ) SvNV(sv);
    return 1;
}

static int wrap_get_A(SV *sv, MAGIC *mg) {
    sv_setnv(sv, (double) A);
    return 1;
}
/* Module initialization function */
XS(boot_example) {
         dXSARGS;
         char *file = __FILE__;
         MAGIC *mg;
         SV *sv;
         newXS("example::foo", wrap_foo, file);

         /* Create a link to the variable A */
         sv = perl_get_sv("example::A",TRUE);
         sv_setnv(sv,A);
         sv_magic(sv,sv,'U',"A",1);
         mg = mg_find(sv,'U');
         mg->mg_virtual = (MGVTBL *) malloc(sizeof(MGVTBL));
         mg->mg_virtual->svt_get = wrap_get_A;
         mg->mg_virtual->svt_set = wrap_set_A;
         mg->mg_virtual->svt_len = 0;
         mg->mg_virtual->svt_clear = 0;
         mg->mg_virtual->svt_free = 0;

         /* Create a constant */
         sv = perl_get_sv("PI",TRUE);
         sv_setnv(sv,PI);
         svREADONLY_on(sv);
         ST(0) = &sv_yes;
         XSRETURN(1);
}
```

A.2 A Python Extension Module

```
/*
 * A simple Python extension module
 */
#include "Python.h"

/* Application specific headers */
#include "example.h"

/* Wrapper for int foo(int a, double b, char *c) */
static
PyObject *wrap_foo(PyObject *self, PyObject *args) {
    PyObject * resultobj;
    int  result;
    int  arg0;
    double  arg1;
    char * arg2;

    if(!PyArg_ParseTuple(args,"ids:foo",&arg0,&arg1,&arg2))
        return NULL;
    result = foo(arg0,arg1,arg2);
    resultobj = Py_BuildValue("i",result);
    return resultobj;
}

/* Wrappers for setting and getting A */
static
PyObject *wrap_A_get(PyObject *self, PyObject *args) {
    if(!PyArg_ParseTuple(args,":A_get"))
        return NULL;
    return Py_BuildValue("d",A);
}

static
PyObject *wrap_A_set(PyObject *self, PyObject *args) {
    double  value;

    if(!PyArg_ParseTuple(args,"d:A_set",&value));
        return NULL;
    A = value;
    return Py_BuildValue("d",A);
}

/* Methods table */
static PyMethodDef exampleMethods[] = {
 { "foo", wrap_foo, 1 },
         { "A_get", wrap_A_get, 1},
         { "A_set", wrap_A_set, 1},
 { NULL, NULL }
};

/* Initialization function */
void initexample() {
 PyObject *m, *d;
 m = Py_InitModule("example", exampleMethods);
 d = PyModule_GetDict(m);

         /* Create a constant */
 PyDict_SetItemString(d,"PI", PyFloat_FromDouble(PI);
}
```

A.3 A Tcl Extension Module

```
/*
 * A simple Tcl extension module
 */

#include <tcl.h>
#include <string.h>

/* Header files from the original application */
#include "example.h"

/* Wrapper for foo(int a, double c, char *) */

int
wrap_foo(ClientData clientData, Tcl_Interp *interp,
         int argc, char *argv[])
{
    int  result;
    int  arg0;
    double  arg1;
    char * arg2;

    if (argc != 4) {
        Tcl_SetResult(interp, "Wrong # args. foo a b c ",TCL_STATIC);
        return TCL_ERROR;
    }
    arg0 = (int) atol(argv[1]);
    arg1 = (double) atof(argv[2]);
    arg2 = argv[3];
    result = foo(arg0,arg1,arg2);
    sprintf(interp->result,"%d", result);
    return TCL_OK;
}
```

```
/* Module initialization function */

int Example_Init(Tcl_Interp *interp) {
    if (!interp)
        return TCL_ERROR;
    Tcl_CreateCommand(interp, "foo", wrap_foo, (ClientData) NULL,
                     (Tcl_CmdDeleteProc *) NULL);

    /* Link to the global variable */
    Tcl_LinkVar(interp, "A", (char *) &A, TCL_LINK_DOUBLE);

    /* Create a constant as a read only variable */
    {
       static double wrap_PI = PI;
       Tcl_LinkVar(interp, "PI", (char *) &wrap_PI,
                   TCL_LINK_DOUBLE | TCL_LINK_READ_ONLY);
    }
    return TCL_OK;
}
```

APPENDIX B

SWIG DIRECTIVES

SWIG has a number of special directives that are used to guide the interface generation process. This section briefly describes a number of the most common directives.

B.1 Code Insertion

The output files of SWIG are divided into three sections. A header section section contains header files and other support code, a wrapper section contains the wrapper code generated by SWIG, a the initialization section contains the module initialization function. The following directives can be used to insert supporting C/C++ code into the output file generated by SWIG.

`%{ ... %}`

All of the code enclosed in the braces is copied verbatim into the header section of the output file. This is typically used to include header files and other support code. The SWIG parser ignores all of the included code.

`%init %{ ... %}`

Copies the code enclosed in the braces into the module initialization function. The included code is ignored by the SWIG parser.

`%inline %{ ... %}`

Copies the code enclosed in the braces into the header section of the output file, but also passes the enclosed code to the SWIG parser. This directive is described in more detail in Section 5.3.

`%wrapper %{ ... %}`

Copies the code enclosed in the braces into the the wrapper section of the output file. The parser ignores the contents of the included code.

B.2 File Inclusion

SWIG interfaces can be broken up into multiple files and assembled to form an interface. The following directives are used to include files and gather interface building information.

`%include filename`

Inserts the contents of a file into the current interface specification. The included file may be a special SWIG interface file, a C/C++ header file, or a C/C++ source file.

`%extern filename`

Loads a file and extracts type information (including structure and class definitions). However, no scripting language wrappers are generated. This directive is primarily use to provide SWIG with information about the underlying C/C++ program without generating any wrapper code.

`%import filename`

Loads information about the contents of another SWIG generated module without generating any wrapper code. This directive is used when working with collection of modules and in cases where one module may depend on the contents of another module.

B.3 Renaming

Sometimes the name of a C function conflicts with a keyword or built-in function in the target scripting language. To resolve these conflicts, the name of functions and variables used in the scripting interface can be changed using the following directives.

`%name(newname) decl`

This directive can be placed in front of any C/C++ declaration to change the name used in the scripting language interface.

`%rename oldname newname`

This directive performs a global renaming operation. It operates like the `%name` directive except that it applies to all occurrences of the old name.

B.4 Access Control

The following directives can be used to control the access to global variables and structure members. Using these directives, a user can be prevented from modifying data from the scripting interface.

`%readonly`

This enables read-only mode. All global variables and data members of classes will be processed so that they can not be modified from the scripting language interpreter. This mode stays in effect until it is explicitly disabled.

`%readwrite`

This directive disables the read-only mode.

B.5 Customization

The following directives are used to customize SWIG's processing. More detailed descriptions about these directives are provided later in this chapter.

`%except(lang) { ... }`

This directive defines a new exception handler as described in section 4.11.

`%typemap(lang,method) datatype { ... }`

Defines a new typemap as described in section 4.10.

`%apply datatype { typelist };`

Applies a typemap to a list of new datatypes as described in section 4.10.1.

B.6 Documentation

The following directives are used to control the documentation generation capability of SWIG.

`%title "text"`

Sets the title of the documentation file.

`%section "text"`

Starts a new documentation section.

`%subsection "text"`

Starts a new documentation subsection.

`%subsubsection "text"`

Starts a new documentation subsubsection.

`%disabledoc`

Disables the documentation system.

`%enabledoc`

Enables the documentation system.

B.7 Miscellaneous Directives

`%module name`

Sets the name of the SWIG extension module. Usually this directive appears once at the beginning of an interface description.

`%native(name) function;`

This directive can be used to add an existing scripting language wrapper function to a SWIG interface. `name` is the name of the scripting language command to be created and `function` is the name of the wrapper function.

`%new decl`

This directive gives a hint to the compiler that a function is returning newly allocated memory. SWIG can sometimes use this to eliminate memory leaks.

`%addmethods classname { ... }`

Adds new methods to C++ classes and C structures as described in section 4.9.4.

APPENDIX C

USER SURVEY

C.1 Languages and Operating Systems

1. What scripting languages do you use with SWIG? (check all that apply)

    - Guile, Perl, Python, Tcl, Other

2. What compiled languages do you use with SWIG? (check all that apply)

    - ANSI C, C++, Objective-C, Fortran, Other

3. What operating systems are you using with SWIG? (check all that apply)

    - Linux, Solaris, SunOS, Irix, HPUX, AIX, Digital Unix, BSD, Macintosh, Windows-NT, Windows-95, Windows-3.1, Other

C.2 SWIG Usage

4. What Version of SWIG are you using?

5. Approximately how many functions do you typically include in your SWIG interfaces (This includes C functions, C++ member functions, etc...)

    - 0-49, 50-99, 100-249, 250-499, 500-999, 1000 up

6. What kind of input do you usually give to SWIG?

    - Separate interface files, Header files, Both

7. What SWIG features do you use regularly? (check all that apply)

    - File inclusion (%include), Exception handling (%except), Shadow classes, Typemaps, Class extension (%addmethods), Documentation generation, The %import directive, The %apply directive, The %name directive, Runtime libraries

8. Do you use any of the following SWIG library files?

   - pointer.i, typemaps.i, exception.i, constraint.i

9. What SWIG documentation format do you use?

   - ASCII, HTML, LaTeX, None

10. How is SWIG installed on your system?

    - In your own directory, In a system directory

11. How do you run SWIG?

    - Directly from the command line, Using a Makefile, From a development environment

12. How often do you use SWIG?

    - Daily, Weekly, Monthly, Rarely

13. Have you ever used SWIG with more than one scripting language?

14. Have you ever modified the wrapper code generated by SWIG?

15. Have you ever modified the SWIG source code or written a new language module?

16. How you do typically build scripting extensions?

    - Dynamic Loading, Static linking

17. Have you ever generated an extension module that was too large to be compiled by your C/C++ compiler?

18. Approximately how long does it take to build a SWIG extension on your machine (running SWIG and compiling the wrapper code with the C/C++ compiler)?

    - 0 - 30 seconds, 30 - 60 seconds, 1 - 2 minutes, 2 - 5 minutes, 5 - 10 minutes, More than 10 minutes

C.3 Evaluation

These questions ask you to evaluate various aspects of SWIG and statements about its use. Score each question on a scale of 1-5 with (1 = Disagree) and (5 = Agree).

19. SWIG is easy to install.

20. It was easy to build your first SWIG example.

21. In practice, SWIG is easy to use.

22. The scripting interfaces created by SWIG are easy to use.

23. How would you rate the quality and accuracy of the SWIG documentation?

24. (Question withdrawn).

25. SWIG generated modules can be quickly compiled.

26. SWIG requires no modifications to the underlying C/C++ code.

27. Parsing ANSI C/C++ declarations makes SWIG easier to use (as opposed to using a special interface definition language).

28. SWIG allows you to build scripting interfaces without having to know all of the gory underlying details.

29. The documentation files created by SWIG are useful.

30. Typemaps are an effective customization mechanism.

31. SWIG/Scripting has had a positive impact on your programming projects.

32. Using SWIG is fun.

C.4 Future Features

33. Which one of the following features would you most like to see? (check only one)

    - Support for Fortran, Better C/C++ parsing, Support for overloaded functions, Optimized output, An extension mechanism, Support for Java, More library files, Better support for arrays

34. If you use C++, do you use any of the following features

    - Templates, Namespaces, Exceptions, Operator overloading, Standard template library (STL), Smart pointers, Expression templates, Other (please specify)

C.5 User Profile

35. How long have you been using SWIG?

    - 0-6 months, 6-12 months, 12-18 months, 18-24 months, > 24 months

36. How long have you been programming?

    - 0-5 years, 5-10 years, 10-15 years, 15-20 years, > 20 years

37. How would you characterize your work environment?

    - Commercial software development, Academic, Government, Industrial Research and Development, Self employed

38. How do you or your organization use SWIG?

    - Personal use, In-house application development, Software testing and debugging, Research and development projects, Rapid prototyping, Commercial software development

39. How did you hear about SWIG?

    - From an article, At a conference, USENET, From a search engine, From a colleague, From Dave

40. Do you subscribe to the SWIG mailing list?

41. What other software packages, libraries, and tools have you used?

- Java, CORBA, COM, ILU, Visual Basic, Other scripting tools, Purify, Make, Revision control, Configuration tools, MATLAB, Mathematica, etc., Database packages, MPI, Threads, OpenGL

42. Did you use any scripting languages before using SWIG?

43. Do you work on scientific applications?

44. Have you ever written a graphical user interface?

45. Have you ever written a network application? (sockets, RPC, CGI scripts, etc...)

46. Have you ever been a system administrator?

47. Do you consider yourself to be a professional software engineer (i.e. your primary job is software development).

48. Do you have a degree in computer science?

C.6 Comments

49. What do you like about SWIG?

50. What limitations have you encountered?

51. Can I quote you?

52. How would you improve SWIG?

53. What kinds of applications are you developing with SWIG?

54. Do you have any comments about this survey?

APPENDIX D

SOFTWARE AVAILABILITY

SWIG is freely available and can be downloaded at

`ftp.cs.utah.edu/pub/beazley/SWIG`.

SWIG can also be found on a variety of software distributions including FreeBSD and certain Linux distributions. A SWIG web-site is also available at `www.swig.org`.

The following web-pages contain information about the Perl, Python, and Tcl scripting languages.

- `www.perl.org`
- `www.python.org`
- `www.scriptics.com`

Information about the SPaSM molecular dynamics code described in chapter 7 is available at `bifrost.lanl.gov/MD/MD.html`.

REFERENCES

[1] ADLER, R. M. Emerging standards for component software. *IEEE Computer 28*, 3 (Mar. 1995), 68–77.

[2] ALEXANDER, P., AND GLADDEN, L. How to create an X-window interface to Gnuplot and Fortran programs using the Tcl/Tk toolkit. *Computers in Physics 9*, 1 (Jan. 1995), 57-64.

[3] ALLEN, M., AND TILDESLEY, D. *Computer Simulations of Liquids*. Clarendon Press, Oxford, 1987.

[4] BALAY, S., GROPP, W. D., MCINNES, L. C., AND SMITH, B. F. PETSc 2.0 users manual. Tech. Rep. ANL-95/11 - Revision 2.0.22, Argonne National Laboratory, 1998.

[5] BEAZLEY, D. SWIG : An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th Annual Tcl/Tk Workshop '96, July 10-13, 1996, Monterey, CA* (Berkeley, CA, July 1996), USENIX, pp. 129–139.

[6] BEAZLEY, D. Using SWIG to control, prototype, and debug c programs with Python. In *Proceedings of the 4th International Python Conference, June 4-6, 1996, Livermore, CA* (Reston, VA, June 1996), CNRI/PSA.

[7] BEAZLEY, D. Feeding a large-scale physics application to python. In *Proceedings of the 6th International Python Conference, Oct. 21-28, 1997, San Jose, CA* (Reston, VA, Oct. 1997), CNRI/USENIX, pp. 21–28.

[8] BEAZLEY, D. SWIG and automated C/C++ scripting extensions. *Dr. Dobb's Journal*, 282 (Feb. 1998), 30–36.

[9] BEAZLEY, D. SWIG users manual. Tech. Rep. UUCS-98-012, University of Utah, 1998.

[10] BEAZLEY, D., AND LOMDAHL, P. Message-passing multi-cell molecular dynamics on the Connection Machine 5. *Parallel Computing 20* (1994), 173–195.

[11] BEAZLEY, D., AND LOMDAHL, P. Lightweight computational steering of very large scale molecular dynamics simulations. In *Proceedings of Supercomputing'96, Nov. 17-22, 1996, Pittsburgh, PA* (Los Alamitos, CA, Nov. 1996), IEEE Computer Society. Published on CD-ROM.

[12] BEAZLEY, D., AND LOMDAHL, P. A practical approach to portability and performance problems on massively parallel supercomputers. In *Debugging and Performance Tuning for Parallel Computing Systems* (Los Alamitos, CA, 1996),

IEEE Computer Society, pp. 337–351.

[13] Beazley, D., and Lomdahl, P. Controlling the data glut in large-scale molecular dynamics simulations. *Computers in Physics 11*, 3 (June 1997), 230–238.

[14] Beazley, D., Lomdahl, P., Jensen, N., Giles, R., and Tamayo, P. Parallel algorithms for short-range molecular dynamics. In *Annual Reviews in Computational Physics* (1995), vol. 3, World Scientific, pp. 119–175.

[15] Beazley, D., Lomdahl, P., Tamayo, P., and Jensen, N. A high performance communications and memory caching scheme for molecular dynamics on the CM-5. In *Proceedings of IPPS'94, Apr. 26-29, 1994, Cancun, Mexico* (Los Alamitos, CA, 1994), IEEE Computer Society, pp. 800–809.

[16] Booch, G. *Object-Oriented Analysis and Design*, 2nd ed. Benhamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[17] Brooks, F. P. *The Mythical Man-Month*, anniversary ed. Addison-Wesley, Reading, MA, 1995.

[18] Bruaset, A. M., and Langtangen, H. P. Object-oriented design of preconditioned iterative methods in Diffpack. *ACM Transactions on Mathematical Software 23*, 1 (Mar. 1997), 50–80.

[19] Brydon, D. *The Method of Patches for Solving Stiff Nonlinear Differential Equations*. PhD thesis, University of Texas at Austin, 1998.

[20] Bryson, S. The data glut revisted. *Computers in Physics 9*, 5 (Sept. 1995), 525–530.

[21] Burnett, M., Hossli, R., Pulliam, T., VanVoorst, B., and Yang, X. Toward visual programming languages for steering scientific computations. *IEEE Computational Science and Engineering 1*, 4 (1994), 44–62.

[22] Char, B. W., Geddes, K. O., Gonnet, G. H., Leong, B., Monagan, M. B., and Watt, S. M. *Maple Library V Reference Manual*. Springer-Verlag, New York, 1991.

[23] Cox, B. J. *Object-Oriented Programming - an Evolutionary Approach*. Addison-Wesley, Reading, MA, 1986.

[24] Cushing, J., Maier, D., Feller, D., and De Vaney, M. Computational proxies: Modeling scientific applications in object databases. In *Seventh International Working Conference on Scientific and Statistical Database Management* (Los Alamitos, CA, Sept. 1994), IEEE Computer Society, pp. 196–206.

[25] Cutting, D., Janssen, B., Spreitzer, M., and Wymore, F. *ILU Reference Manual*. Xerox Palo Alto Research Center, Dec. 1993. Accessible at `ftp://-ftp.parc.xerox.com/pub/ilu/ilu.html`.

[26] DAZZO, G. How to maintain large scientific programs. *Computers in Physics 2*, 1 (Jan. 1988), 52-56.

[27] DEFANTI, T., FOSTER, I., PAPKA, M., STEVENS, R., AND KUHFUSS, T. Overview of the I-WAY : Wide-area visual supercomputing. *Int. Journal of Supercomputing Applications 10*, 2 (Spring 1996), 123 131.

[28] DINGLE, A., AND HILDEBRANDT, T. Improving C++ performance using temporaries. *IEEE Computer 31*, 3 (Mar. 1998), 31-41.

[29] DUBOIS, P. *Object Technology for Scientific Computing.* Prentice Hall PTR, Upper Saddle River, NJ, 1997.

[30] DUBOIS, P., HINSEN, K., AND HUGUNIN, J. Numerical python. *Computers in Physics 10*, 3 (May 1996), 262-267.

[31] DUBOIS, P. F. Object-oriented programming creates a software revolution. *Computers in Physics 5*, 6 (Nov. 1991), 568 570.

[32] DUBOIS, P. F. Making applications programmable. *Computers in Physics 8*, 1 (Jan. 1994), 70-73.

[33] DUBOIS, P. F. The future of scientific programming. *Computers in Physics 11*, 2 (Mar. 1997), 168-173.

[34] ELLIS, M. A., AND STROUSTRUP, B. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1992.

[35] FANTI, T. D., AND ET. AL. Special issue on visualization in scientific computing. *Computer Graphics 21*, 6 (Nov. 1987).

[36] FAYAD, M., AND SCHMIDT, D. C. Object-oriented application frameworks. *Communications of the ACM 40*, 10 (Oct. 1997), 32 38.

[37] FERGUSON, P., HUMPHREY, W., KHAJENOORI, S., MACKE, S., AND MATVYA, A. Results of applying the personal software process. *IEEE Computer 30*, 5 (May 1997), 24-31.

[38] FLANAGAN, D. *Java in a Nutshell: a desktop quick reference*, 2nd ed. A Nutshell handbook. O'Reilly & Associates, Inc., Sebastopol, CA, 1997.

[39] FLY, C. Grammar-based rapid application development (GRAD). In *5th International Python Conference, Nov. 4-5, 1997, Washington, DC* (Reston, VA, Nov. 1996), CNRI.

[40] FOSTER, I., AND KESSELMAN, C. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing 11*, 2 (Summer 1997), 115-128.

[41] FRANZ, M. Dynamic linking of software components. *IEEE Computer 30*, 3 (Mar. 1997), 74-81.

[42] Friedman, D. P., Wand, M., and Haynes, C. T. *Essentials of Programming Languages.* McGraw Hill, New York, 1992.

[43] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns.* Addison-Wesley, Reading, MA, 1995.

[44] Geist, II, G. A., Kohl, J. A., and Papadopoulos, P. M. CUMULVS: Providing fault tolerance, visualization, and steering of parallel applications. *The International Journal of Supercomputer Applications and High Performance Computing 11*, 3 (Fall 1997), 224–235.

[45] Glass, R. *Software Runaways : Lessons Learned from Massive Software Project Failures.* PTR Prentice Hall, Upper Saddle River, NJ, 1998.

[46] Goldberg, A., and Robson, D. *Smalltalk-80 : the language and its implementation.* Addison-Wesley, 1983.

[47] Grimsaw, A. S., and Wulf, W. A. The Legion vision of a wolrdwide virtual computer. *Communications of the ACM 40*, 1 (Jan. 1997), 39–45.

[48] Gropp, W., Lusk, E., and Skjellum, A. *Using MPI : Portable Parallel Programming with the Message-Passing Interface.* The MIT Press, Cambridge, MA, 1996.

[49] Gu, W., Vetter, J., and Schwan, K. An annotated bibliography of interactive program steering. *ACM SIGPLAN Notices 29*, 9 (Sept. 1994), 140–148.

[50] Haney, S. W. Is C++ fast enough for scientific computing? *Computers in Physics 8*, 6 (Nov. 1994), 690–694.

[51] Haney, S. W. Beating the abstraction penalty in C++ using expression templates. *Computers in Physics 10*, 6 (Nov. 1996), 552–557.

[52] Haney, S. W., and Crotinger, J. C++ proves useful in writing a tokamak systems code. *Computers in Physics 6*, 5 (Sept. 1992), 450–455.

[53] Hanselman, D., and Littlefield, B. *MATLAB Version 5 Users Guide.* Prentice Hall, Upper Saddle River, NJ, 1996.

[54] Heidrich, W., and Slusallek, P. Automatic generation of Tcl bindings for C and C++ libraries. In *Proceedings of the Tcl/Tk Workshop, July 6-8, 1995, Toronto, ON* (Berkeley, CA, July 1995), USENIX, pp. 85–94.

[55] Hinsen, K. The molecular modeling toolkit: A case study of a large scientific application in python. In *Proceedings of the 6th International Python Conference, Oct. 14-17, 1997, San Jose, CA.* (Reston, VA, Oct. 1997), USENIX/CNRI, pp. 29–35.

[56] Hipp, R. *Tcl/Tk Tools.* O'Reilly & Associates, Sebastopol, CA, 1997, ch. 14, pp. 511–534.

[57] Holian, B., Hammerberg, J., and Lomdahl, P. The birth of dislocations in

shock waves and high-speed friction. *Journal of Computer-Aided Materials Design* (1998). To appear.

[58] HOLIAN, B., AND LOMDAHL, P. Plasticity induced by shock waves in nonequilibrium molecular-dynamics simulations. Tech. Rep. LA-UR 98-702, Los Alamos National Laboratory, 1998. To appear in *Science*.

[59] JABLONOWSKI, D., BRUNER, J., BLISS, B., AND HABER, R. VASE: The visualization and application steering environment. In *Proceedings of Supercomputing '93, Nov. 15-19, 1993, Portland, OR* (1993), IEEE Computer Society Press, pp. 560 – 569.

[60] JANSSEN, B., AND SPREITZER, M. ILU : Inter-language unification via object modules. In *Proceedings of OOPSLA 94 Workshop on Multi-Language Object Models* (1994), ACM.

[61] KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language*, 2nd ed. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[62] LEVINE, J., MASON, T., AND BROWN, D. *Lex and yacc.* O'Reilly and Associates, Sebastopol, CA, 1992.

[63] LIBES, D. *Exploring Expect.* O'Reilly and Associates, Sebastopol, CA, 1995.

[64] LOMDAHL, P., TAMAYO, P., JENSEN, N., AND BEAZLEY, D. 50 gflops molecular dynamics on the CM-5. In *Proceedings of Supercomputing 93, Nov. 15-19, 1993, Portland, OR* (Los Alamitos, CA, 1993), IEEE Computer Society, pp. 520–527.

[65] LORD, T. An anatomy of guile: The interface to Tcl/Tk. In *Proceedings of the Tcl/Tk Workshop, July 6–8, 1995, Toronto, ON* (Berkeley, CA, July 1995), USENIX Association, pp. 95–114.

[66] LUTZ, M. *Programming Python.* O'Reilly and Associates, Sebastopol, CA, 1996.

[67] MARTIN, K. Automated wrapping of a C++ class library into Tcl. In *4th Annual Tcl/Tk Workshop '96, July 10–13, 1996. Monterey, CA* (Berkeley, CA, July 1996), USENIX, pp. 141–148.

[68] THE MATHWORKS, INC. *MATLAB : External Interface Guide*, 1993.

[69] MOSER, S., AND NIERSTRASZ, O. The effect of object-oriented frameworks on developer productivity. *IEEE Computer* (Sept. 1996), 45–51.

[70] MOWBRAY, T., AND MALVEAU, R. *CORBA Design Patterns.* John Wiley & Sons, Inc., New York, 1997.

[71] MUNRO, D. H. Using the Yorick interpreted language. *Computers in Physics 9*, 6 (Nov. 1995), 609–615.

[72] MURRAY, J. D., AND VANRYPER, W. *Encyclopedia of Graphics File Formats.* O'Reilly & Associates, Inc., Sebastopol, CA, 1994.

[73] Myers, C. The dynamics of localized coherent structures and the role of adaptive software in multiscale modeling. In *Proceedings of the IMA Workshop on Structured Adaptive Mesh Refinement Grid Methods* (1997), Springer-Verlag. In press.

[74] Object Management Group. The Common Object Request Broker: Architecture and specification. Accessible at `ftp://omg.org/pub/CORBA`, Dec. 1993.

[75] OpenGL Architecture Review Board. *OpenGL Reference Manual: The Official Reference Document for OpenGL, Release 1.* Addison-Wesley, Reading, MA, 1993.

[76] Ousterhout, J. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, MA, 1994.

[77] Ousterhout, J. Scripting : Higher level programming for the 21st century. *IEEE Computer 31*, 3 (Mar. 1998), 23–30.

[78] Pancake, C., and Cook, C. What users need in parallel tool support : Survey results and analysis. In *Proc. Scalable High Performance Computing Conference* (1994), pp. 40–47.

[79] Parker, S., Beazley, D., and Johnson, C. Computational steering software systems and strategies. *IEEE Computational Science and Engineering 4*, 4 (1997), 50–59.

[80] Parker, S., Weinstein, D., and Johnson, C. The SCIRun computational steering software system. In *Modern Software Tools in Scientific Computing*, E. Arge, A. Bruaset, and H. Langtangen, Eds. Birkhauser Press, 1997, pp. 1–44.

[81] Quinlan, D. A++/P++ user manual. Tech. Rep. LA-UR 95-3273, Los Alamos National Laboratory, 1995.

[82] Reed, B. View from X-DO–the challenge of ASCI. *Xwindows 5*, 4 (Winter 1996), 3. X-division newsletter at Los Alamos National Laboratory.

[83] Research Systems, Inc. *Using IDL, Version 5.0*, 1997.

[84] Reynders, J., Hinker, P., Cummings, J., Atlas, S., Banerjee, S., Humphrey, W., Karmesin, S., Keahey, K., Srikant, M., and Tholburn, M. POOMA. In *Parallel Programming Using C++*, G. Wilson and P. Lu, Eds., Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, 1996, pp. 547–588.

[85] Rice, J. Future scientific software systems. *IEEE Computational Science and Engineering* (Apr. 1997), 44–48.

[86] Robison, A. D. C++ gets faster for scientific computing. *Computers in Physics 10*, 5 (Sept. 1996), 458–462.

[87] Rogerson, D. *Inside COM : Microsoft's Component Object Model.* Microsoft Press, Redmond, WA, 1997.

[88] Romer, T. H., Lee, D., Voelker, G. M., Wolman, A., Wong, W. A.,

BAER, J.-L., BERSHAD, B. N., AND LEVY, H. M. The structure and performance of interpreters. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, Oct. 1996), ACM Press, pp. 150–159.

[89] SCHROEDER, W., MARTIN, K., AND LORENSEN, B. *The Visualization Toolkit.* Prentice Hall PTR, Upper Saddle River, NJ, 1996.

[90] SHAW, M., AND GARLAN, D. *Software Architecture. Perspectives on an Emerging Discipline.* Prentice Hall, Upper Saddle River, NJ, 1996.

[91] SRINIVASAN, S. *Advanced Perl Programming.* O'Reilly and Associates, Sebastopol, CA, 1997.

[92] STEVENS, R., WOODWARD, P., DEFANTI, T., AND CATLETT, C. From the I-WAY to the national technology grid. *Communications of the ACM 40*, 11 (Nov. 1997), 51–60.

[93] STEVENS, W. R. *UNIX Network Programming.* PTR Prentice Hall, Englewood Cliffs, NJ, 1990.

[94] SUSSMAN, G. J. *LISP, Programming and Implementation.* Cambridge University Press, London, 1982.

[95] VELDHUIZEN, T. Expression templates. *C++ Report 7*, 5 (June 1995), 26–31.

[96] VELDHUIZEN, T. Scientific computing: C++ versus Fortran: C++ has more than caught up. *Dr. Dobb's Journal of Software Tools 22*, 11 (Nov. 1997), 34, 36–38, 91.

[97] VETTER, J., EISENHAUER, G., GU, W., KINDLER, T., SCHWAN, K., AND SILVA, D. Opportunities and tools for highly interactive distributed and parallel computing. In *Proceedings of the Workshop On Debugging and Tuning for Parallel Computing Systems* (Oct. 1994), pp. 139–142.

[98] VETTER, J., AND SCHWAN, K. Progress: A toolkit for interactive program steering. In *Proceedings of the 24th Annual Conference of International Conference on Parallel Processing* (1995), pp. 139 – 142.

[99] VETTER, J., AND SCHWAN, K. Models for computational steering. In *Proceedings of the Third International Conference on Configurable Distributed Systems* (1996).

[100] VETTER, J., AND SCHWAN, K. High performance computational steering of physical simulations. In *Proceedings of the 11th International Parallel Processing Symposium, Apr. 1-5, 1997, Geneva, Switzerland* (Los Alamitos, CA, Apr. 1997), IEEE Computer Society.

[101] WALL, L., CHRISTIANSEN, T., AND SCHWARTZ, R. *Programming Perl*, 2nd ed. O'Reilly and Associates, Sebastopol, CA, 1996.

[102] WARREN, M., GERMANN, T., LOMDAHL, P., BEAZLEY, D., AND SALMON, J. Avalon: An Alpha/Linux cluster achieves 10 gflops for $150k. In *Proceedings of*

Supercomputing 98 (Los Alamitos, CA, 1998), IEEE Computer Society. To appear.

[103] WARREN, M., SALMON, J., BECKER, D., GODA, M., STERLING, T., AND WINCKELMANS, G. Pentium Pro inside: I. a treecode at 430 gigaflops on ASCI red, II. price/performance of $50/mflop on Loki and Hyglac. In *Proceedings of Supercomputing 97, Nov. 15-21, 1997, San Jose, CA* (Los Alamitos, CA, 1997), IEEE Computer Society.

[104] WATTERS, A., VAN ROSSUM, G., AND AHLSTROM, J. C. *Internet Programming with Python.* MT Books, New York, 1996.

[105] WELCH, B. *Practical Programming in Tcl and Tk*, 2nd ed. Prentice Hall PTR, Upper Saddle River, NJ, 1997.

[106] WETHERALL, D., AND LINDBLAD, C. J. Extending Tcl for dynamic object-oriented programming. In *Proceedings of the Tcl/Tk Workshop, July 6–8, 1995, Toronto, ON* (Berkeley, CA, July 1995), USENIX, pp. 173–182.

[107] WINFIELD, A. J. A virtual laboratory notebook for simulation models. In *Pacific Symposium on Biocomputing '98* (Jan. 1998), R. B. Altman, D. K. Keith, and L. Hunter, Eds., vol. 3, World Scientific Pub Co., pp. 177–188. PSB'98 electronic proceedings availiable at http://www.cgl.ucsf.edu/psb/psb98/.

[108] WOLFRAM, S. *Mathematica: A System for Doing Mathematics By Computer*, 2nd ed. Addison-Wesley, Reading, MA, 1991.

[109] YOURDON, E. *Decline and Fall of the American Programmer.* PTR Prentice Hall, Englewood Cliffs, NJ, 1993.

[110] ZHOU, S., BEAZLEY, D., AND LOMDAHL, P. Large-scale molecular dynamics simulations of three-dimensional fracture. *Physical Review Letters 78*, 3 (1997), 479–482.

[111] ZHOU, S., PRESTON, D., LOMDAHL, P., AND BEAZLEY, D. Large-scale molecular dynamics simulations of dislocation intersection in copper. *Science 279* (Mar. 1998), 1525–1527.