The following paper was originally published in the
Proceedings of the Fourth USENIX Tcl/Tk Workshop
Monterey, CA, July 10-13, 1996

SWIG : An Easy to Use Tool For Integrating Scripting Languages with C and C++

David M. Beazley
University of Utah
Salt Lake City, Utah 84112

SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++

David M. Beazley
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112
beazley@cs.utah.edu

Abstract

I present SWIG (Simplified Wrapper and Interface Generator), a program development tool that automatically generates the bindings between C/C++ code and common scripting languages including Tcl, Python, Perl and Guile. SWIG supports most C/C++ datatypes including pointers, structures, and classes. Unlike many other approaches, SWIG uses ANSI C/C++ declarations and requires the user to make virtually no modifications to the underlying C code. In addition, SWIG automatically produces documentation in HTML, LaTeX, or ASCII format. SWIG has been primarily designed for scientists, engineers, and application developers who would like to use scripting languages with their C/C++ programs without worrying about the underlying implementation details of each language or using a complicated software development tool. This paper concentrates on SWIG's use with Tcl/Tk.

1 Introduction

SWIG (Simplified Wrapper and Interface Generator) is a software development tool that I never intended to develop. At the time, I was trying to add a data analysis and visualization capability to a molecular dynamics (MD) code I had helped develop for massively parallel supercomputers at Los Alamos National Laboratory [Beazley, Lomdahl]. I wanted to provide a simple, yet flexible user interface that could be used to glue various code modules together and an extensible scripting language seemed like an ideal solution. Unfortunately there were constraints. First, I didn't want to hack up 4-years of code development trying to fit our MD code into yet another interface scheme (having done so several times already). Secondly, this code was routinely run on systems ranging from Connection Machines and Crays to workstations and I didn't want to depend on any one interface language—out of fear that it might not be supported on all of these platforms. Finally, the users were constantly adding new code and making modifications. I needed a flexible, yet easy to use system that did not get in the way of the physicists.

SWIG is my solution to this problem. Simply stated, SWIG automatically generates all of the code needed to bind C/C++ functions with scripting languages using only a simple input file containing C function and variable declarations. At first, I supported a scripting language I had developed specifically for use on massively parallel systems. Later I decided to rewrite SWIG in C++ and extend it to support Tcl, Python, Perl, Guile and other languages that interested me. I also added more data-types, support for pointers, C++ classes, documentation generation, and a few other features.

This paper provides a brief overview of SWIG with a particular emphasis on Tcl/Tk. However, the reader should remain aware that SWIG works equally well with Perl and other languages. It is not my intent to provide a tutorial or a user's guide, but rather to show how SWIG can be used to do interesting things such as adding Tcl/Tk interfaces to existing C applications, quickly debugging and prototyping C code, and building interface-language-independent C applications.

2 Tcl and Wrapper Functions

In order to add a new C or C++ function to Tcl, it is necessary to write a special "wrapper" function that parses the function arguments presented as ASCII strings by the Tcl interpreter into a representation that can be used to call the C function. For example, if you wanted to add the factorial function to Tcl, a wrapper function might look like the following :

```
int wrap_fact(ClientData clientData,
              Tcl_Interp *interp,
              int argc, char *argv[]) {
    int result;
    int arg0;
    if (argc != 2) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    arg0 = atoi(argv[1]);
    result = fact(arg0);
    sprintf(interp->result,"%d",result);
    return TCL_OK;
}
```

In addition to writing the wrapper function, a user will also need to write code to add this function to the Tcl interpreter. In the case of Tcl 7.5, this could be done by writing an initialization function to be called when the extension is loaded dynamically. While writing a wrapper function usually is not too difficult, the process quickly becomes tedious and error prone as the number of functions increases. Therefore, automated approaches for producing wrapper functions are appealing–especially when working with a large number of C functions or with C++ (in which case the wrapper code tends to get more complicated).

3 Prior Work

The idea of automatically generating wrapper code is certainly not new. Some efforts such as Itcl++, Object Tcl, or the XS language included with Perl5, provide a mechanism for generating wrapper code, but require the user to provide detailed specifications, type conversion rules, or use a specialized syntax [Heidrich, Wetherall, Perl5]. Large packages such as the Visualization Toolkit (vtk) may use their own C/C++ translators, but these almost always tend to be somewhat special purpose (in fact, SWIG started out in this manner) [vtk]. If supporting multiple languages is the ultimate goal, a programmer might consider a package such as ILU [Janssen]. Unfortunately, this requires the user to provide specifications in IDL–a process which is unappealing to many users. SWIG is not necessarily intended to compete with these approaches, but rather is designed to be a no-nonsense tool that scientists and engineers can use to easily add Tcl and other scripting languages to their own applications. SWIG is also very different than Embedded Tk (ET) which also aims to simplify code development [ET]. Unlike ET, SWIG is designed to integrate C functions into Tcl/Tk as opposed to integrating Tcl/Tk into C programs.

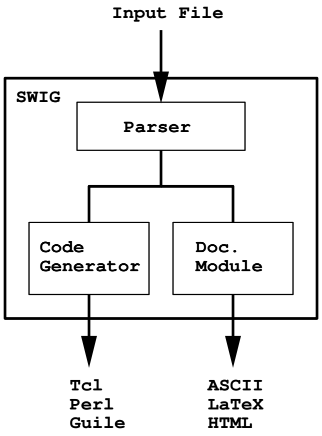

Figure 1: SWIG organization.

4 A Quick Tour of SWIG

4.1 Organization

Figure 1 shows the structure of SWIG. At the core is a YACC parser for reading input files along with some utility functions. To generate code, the parser calls about a dozen functions from a generic language class to do things like write a wrapper function, link a variable, wrap a C++ member function, etc... Each target language is implemented as a C++ class containing the functions that emit the resulting C code. If an "empty" language definition is given to SWIG, it will produce no output. Thus, each language class can be implemented in almost any manner. The documentation system is implemented in a similar manner and can currently produce ASCII, LaTeX, or HTML output. As output, SWIG produces a C file that should be compiled and linked with the rest of the code and a documentation file that can be used for later reference.

4.2 Interface Files

As input, SWIG takes a single input file referred to as an "interface file." This file contains a few SWIG specific directives, but otherwise contains ANSI C function and variable declarations. Unlike the approach in [Heidrich], no type conversion rules are needed and all declarations are made using familiar ANSI C/C++ prototypes. The following code shows an interface file for wrapping a few C file I/O and memory management functions.

```
/* File : file.i */
%module fileio
%{
#include <stdio.h>
%}

FILE *fopen(char *filename, char *type);
int  fclose(FILE *stream);
typedef unsigned int size_t
size_t  fread(void *ptr, size_t size,
              size_t nobj, FILE *stream);
size_t  fwrite(void *ptr, size_t size,
               size_t nobj,FILE *stream);
void *malloc(size_t nbytes);
void  free(void *);
```

The `%module` directive sets the name of the initialization function. This is optional, but is recommended if building a Tcl 7.5 module. Everything inside the `%{, %}` block is copied directly into the output, allowing the inclusion of header files and additional C code. Afterwards, C/C++ function and variable declarations are listed in any order. Building a new Tcl module is usually as easy as the following :

```
unix > swig -tcl file.i
unix > gcc file_wrap.c -I/usr/local/include
unix > ld -shared file_wrap.o -o Fileio.so
```

4.3 A Tcl Example

Newly added functions work like ordinary Tcl procedures. For example, the following Tcl script copies a file using the binary file I/O functions added in the previous example :

```
proc filecopy {name1 name2} {
    set buffer [malloc 8192];
    set f1 [fopen $name1 r];
    set f2 [fopen $name2 w];
    set nbytes [fread $buffer 1 8192 $f1];
    while {$nbytes > 0} {
        fwrite $buffer 1 $nbytes $f2;
        set nbytes [fread $buffer 1 8192 $f1];
    }
    fclose $f1;
    fclose $f2;
    free $buffer
}
```

4.4 Datatypes and Pointers

SWIG supports the basic datatypes of `int`, `short`, `long`, `float`, `double`, `char`, and `void` as well as signed and unsigned integers. SWIG also allows derived types such as pointers, structures, and classes, but these are all encoded as pointers. If an unknown type is encountered, SWIG assumes that it is a complex datatype that has been defined earlier. No attempt is made to figure out what data that datatype actually contains or how it should be used. Of course, this this is only possible since SWIG's mapping of complex types into pointers allows them to be handled in a uniform manner. As a result, SWIG does not normally need any sort of type-mapping, but `typedef` can be used to map any of the built-in datatypes to new types if desired.

SWIG encodes pointers as hexadecimal strings with type-information. This type information is used to provide a run-time type checking mechanism. Thus, a typical SWIG pointer looks something like the following :

```
_1008e614_Vector_p
```

If this pointer is passed into a function requiring some other kind of pointer, SWIG will generate a Tcl error and return an error message. The NULL pointer is represented by the string "NULL". The SWIG run-time type checker is saavy to typedefs and the relationship between base classes and derived classes in C++. Thus if a user specifies

```
typedef double Real;
```

the type checker knows that `Real *` and `double *` are equivalent (more on C++ in a minute). From the point of view of other Tcl extensions, SWIG pointers should be seen as special "handles" except that they happen to contain the pointer value and its type.

To some, this approach may seem horribly restrictive (or error prone), but keep in mind that SWIG was primarily designed to work with existing C applications. Since most C programs pass complex datatypes around by reference this technique works remarkably well in practice. Run time type-checking also eliminates most common crashes by catching stupid mistakes such as using a wrong variable name or forgetting the "$" character in a Tcl script. While it is still possible to crash Tcl by forging a SWIG pointer value (or making a call to buggy C code), it is worth emphasizing that existing Tcl extensions may also crash if given an invalid handle.

4.5 Global Variables and Constants

SWIG can install global C variables and constants using Tcl's variable linkage mechanism. Variables may also be declared as "read only" within the Tcl interpreter. The following example shows how variables and constants can be added to Tcl :

```
// SWIG file with variables and constants
%{
%}

// Some global variables
extern  int My_variable;
extern  char *default_path;
extern  double My_double;

// Some constants
#define  PI     3.14159265359
#define  PI_4   PI/4.0
enum colors {red,blue,green};
const int SIZEOF_VECTOR = sizeof(Vector);

// A read only variable
%readonly
extern int Status;
%readwrite
```

4.6 C++ Support

The SWIG parser can handle simple C++ class definitions and supports public inheritance, virtual functions, static functions, constructors and destructors. Currently, C++ translation is performed by politely tranforming C++ code into C code and generating wrappers for the C functions. For example, consider the following SWIG interface file containing a C++ class definition:

```
%module tree
%{
#include "tree.h"
%}

class Tree {
public:
   Tree();
  ~Tree();
   void  insert(char *item);
   int   search(char *item);
   int   remove(char *item);
static void print(Tree *t);
};
```

When translated, the class will be access used the following set of functions (created automatically by SWIG).

```
Tree *new_Tree();
void delete_Tree(Tree *this);
void Tree_insert(Tree *this, char *item);
int  Tree_search(Tree *this, char *item);
int  Tree_remove(Tree *this, char *item);
void Tree_print(Tree *t);
```

All C++ functions wrapped by SWIG explicitly require the `this` pointer as shown. This approach has the advantage of working for all of the target languages. It also makes it easier to pass objects between other C++ functions since every C++ object is simply represented as a SWIG pointer. SWIG does not support function overloading, but overloaded functions can be resolved by renaming them with the SWIG `%name` directive as follows:

```
class List {
public:
               List();
%name(ListMax) List(int maxsize);
...
}
```

The approach used by SWIG is quite different than that used in systems such as Object Tcl or vtk [vtk, Wetherall]. As a result, users of those systems may find it to be confusing. However, It is important to note that the modular design of SWIG allows the user to completely redefine the output behavior of the system. Thus, while the current C++ implementation is quite different than other systems supporting C++, it would be entirely possible write a new SWIG module that wrapped C++ classes into a representation similar to that used by Object Tcl (in fact, in might even be possible to use SWIG to produce the input files used for Object Tcl).

4.7 Multiple Files and Code Reuse

An essential feature of SWIG is its support for multiple files and modules. A SWIG interface file may include another interface file using the "`%include`" directive. Thus, an interface for a large system might be broken up into a collection of smaller modules as shown

```
%module package
%{
#include "package.h"
%}

%include geometry.i
%include memory.i
%include network.i
%include graphics.i
%include physics.i

%include wish.i
```

Common operations can be placed into a SWIG library for use in all applications. For example, the `%include wish.i` directive tells SWIG to include code for the `Tcl_AppInit()` function needed to rebuild the `wish` program. The library can also be used to build modules allowing SWIG to be used with common Tcl extensions such as Expect [Expect]. Of course, the primary use of the library is with large applications such as Open-Inventor which contain hundreds of modules and a substantial class hierarchy [Invent]. In this case a user could use SWIG's include mechanism to selectively pick which modules they wanted to use for a particular problem.

4.8 The Documentation System

SWIG produces documentation in ASCII, LaTeX, or HTML format describing everything that was wrapped. The documentation follows the syntax rules of the target language and can can be further enhanced by adding descriptions in a C/C++ comment immediately following a declaration. These comments may also contain embedded LaTeX or HTML commands. For example:

```
extern size_t fread(void *ptr, size_t size,
                    size_t nobj, FILE *stream);
/* {\tt fread} reads from {\tt stream} into
the array {\tt ptr} at most {\tt nobj} objects
of size {\tt size}.   {\tt fread} returns
the number of objects read. */
```

When output by SWIG and processed by LaTeX, this appears as follows :

`size_t :  fread ptr size nobj stream`

> `fread` reads from `stream` into the array `ptr` at most `nobj` objects of size `size`. `fread` returns the number of objects read.

4.9 Extending the SWIG System

Finally, SWIG itself can be extended by the user to provide new functionality. This is done by modifying an existing or creating a new language class. A typical class must specify the following functions that determine the behavior of the parser output :

```
// File : swigtcl.h
class TCL : public Language {
private:
  // Put private stuff here
public :
  TCL();
  int main(int, char *argv[]);
  void create_function(char *,char *,DataType*,
                       ParmList *);
  void link_variable(char *,char *,DataType *);
  void declare_const(char *,int,char *);
  void initialize(void);
  void headers(void);
  void close(void);
  void usage_var(char *,DataType*,char **);
  void usage_func(char *,DataType*,ParmList*,
                  char **);
  void usage_const(char *,int,char*,char**);
  void set_module(char *);
  void set_init(char *);
};
```

Descriptions of these functions can be found in the SWIG users manual. To build a new version of SWIG, the user only needs to provide the function definitions and a main program which looks something like the following :

```
// SWIG main program

#include "swig.h"
#include "swigtcl.h"

int main(int argc, char **argv) {

  Language *lang;

  lang = new TCL;
  SWIG_main(argc,argv,lang,(Documentation *) 0);

}
```

When linked with a library file, any extensions and modifications can now be used with the SWIG parser. While writing a new language definition is not entirely trivial, it can usually be done by just copying one of the existing modules and modifying it appropriately.

5 Examples

5.1 A C-Tcl Linked List

SWIG can be used to build simple data structures that are usuable in both C and Tcl. The following code shows a SWIG interface file for building a simple linked list.

```
/* File : list.i */
%{
struct Node {
  Node(char *n) {
```

```
    name = new char[strlen(n)+1];
    strcpy(name,n);
    next = 0;
  };
  char  *name;
  Node *next;
};
%}

// Just add struct definition to
// the interface file.

struct Node {
  Node(char *);
  char *name;
  Node *next;
};
```

When used in a Tcl script, we can now create new nodes and access individual members of the `Node` structure. In fact, we can write code to convert between Tcl lists and linked lists entirely in Tcl as shown :

```
# Builds linked list from a Tcl list
proc buildlist {list head} {
    set nitems [llength $list];
    for {set i 0} {$i < $nitems} {incr i -1} {
        set item [lrange $list $i $i]
        set n [new_Node $item]
        Node_set_next $n $head
        set head $n
    }
    return $head
}
# Builds a Tcl list from a linked list
proc get_list {llist} {
    set list {}
    while {$llist != "NULL"} {
        lappend list [Node_name_get $llist]
        set llist [Node_get_next $llist]
    }
    return $list
}
```

When run interactively, we could now use our Tcl functions as follows.

```
% set l {John Anne Mary Jim}
John Anne Mary Jim
% set ll [buildlist $l _0_Node_p]
_1000cab8_Node_p
% get_list $ll
Jim Mary Anne John
% set ll [buildlist {Mike Peter Dave} $ll]
_1000cc38_Node_p
% get_list $ll
Dave Peter Mike Jim Mary Anne John
%
```

Aside from the pointer values, our script acts like any other Tcl script. However, we have built up a real C data structure that could be easily passed to other C functions if needed.

5.2 Using C Data-Structures with Tk

In manner similar to the linked list example, Tcl/Tk can be used to build complex C/C++ data structures. For example, suppose we wanted to interactively build a graph of "Nodes" for use in a C application. A typical interface file might include the following functions:

```
%{
#include "nodes.h"
%}

%include wish
extern Node *new_node();
extern void AddEdge(Node *n1, Node *n2);
```

Within a Tcl/Tk script, loosely based on one to make ball and stick graphs in [Ousterhout], a graph could be built as follows:

```
proc makeNode {x y} {
   global nodeX nodeY nodeP edgeFirst edgeSecond
   set new [.create oval [expr $x-15] \
           [expr $y-15] [expr $x+15] \
           [expr $y+15] -outline black \
           -fill white -tags node]
   set newnode [new_node]
   set nodeX($new) $x
   set nodeY($new) $y
   set nodeP($new) $newnode
   set edgeFirst($new) {}
   set edgeSecond($new) {}
}
proc makeEdge {first second} {
   global nodeX nodeY nodeP edgeFirst edgeSecond
   set x1 $nodeX($first);  set y1 $nodeY($first)
   set x2 $nodeX($second); set y2 $nodeY($second)
   set edge [.c create line $x1 $y1 $x2 $y2 \
             -tags edge}
   .c lower edge
   lappend edgeFirst($first) $edge
   lappend edgeSecond($first) $edge
   AddEdge $nodeP($first) $nodeP($second)
}
```

These functions create Tk canvas items, but also attach a pointer to a C data structure to each one. This is done by maintaining an associative array mapping item identifiers to pointers (with the `nodeP()` array). When a particular "node" is referenced later, we can use this to get its pointer use it in calls to C functions.

5.3 Parsing a C++ Simple Class Hierarchy

As mentioned earlier, SWIG can handle C++ classes and public inheritance. The following example provides a few classes and illustrates how this is accomplished (some code has been ommitted for readability).

```
// A SWIG inheritance example
%module shapes
%{
#include "shapes.h"
%}

class Shape {
private:
  double xc, yc;
public:
  virtual double area() = 0;
  virtual double perimeter() = 0;
  void    set_position(double x, double y);
  void    print_position();
};

class Circle: public Shape {
 private:
  double radius;
 public:
  Circle(double r);
  double area();
  double perimeter();
};

class Square : public Shape {
private:
  double width;
public:
  Square(double w);
  double area();
  double perimeter();
};
```

Now, when wrapped by SWIG [1], we can use our class structure as follows:

```
% set c [new_Circle 4]
_1000ad70_Circle_p
% set s [new_Square 10]
_1000adc0_Square_p
% Shape_area $c
50.26548246400000200
% Shape_area $s
100.00000000000000000
% Shape_set_position $c -5 10
% Circle_print_position $c
xc = -5, yc = 10
%
```

[1] When parsing C++ classes, SWIG throws away everything declared as private, inline code, and alot of the other clutter found in C++ header files. Primarily this is provided only to make it easier to build interfaces from existing C++ header files.

In our example, we have created new `Circle` and `Square` objects, but these can be used interchangably in any functions defined in the `Shape` base class. The SWIG type checker is encoded with the class hierarchy and knows the relationship between the different classes. Thus, while an object of type `Circle` is perfectly acceptable to a function operating on shapes, it would be unacceptable to a function operating only on the `Square` type. As in C++, any functions in the base class can be called in the derived class as shown by the `Circle_print_position` function above.

6 Using SWIG in Real Applications

So far only a few simple toy examples have been presented to illustrate the operation of SWIG in general. This section will describe how SWIG can be used with larger applications.

6.1 Use in Scientific Applications

Many users, especially within the scientific and engineering community, have spent years developing simulation codes. While many of these users appreciate the power that a scripting language can provide, they don't want to completely rewrite their applications or spend all of their time trying to build a user-interface (most users would rather be working on the scientific problem at hand). While SWIG certainly won't do everything, it can dramatically simplify this process.

As an example, the first SWIG application was the SPaSM molecular dynamics code developed at Los Alamos National Laboratory [Beazley, Lomdahl]. This code is currently being used for materials science problems and consists of more than 200 C functions and about 20000 lines of code. In order to use SWIG, only the `main()` function had to be rewritten along with a few other minor modifications. The full user interface is built using a collection of modules for simulation, graphics, memory management, etc... A user may also supply their own interface modules–allowing the code to be easily extended with new functionality capabilities as needed. All of the interface files, containing a few hundred lines of function declarations, are automatically translated into more than 2000 lines of wrapper code at compile time. As a result, many users of the system know how to extend it, but are unaware of the actual wrapping procedure.

After modifying the SPaSM code to use SWIG, most of the C code remained unchanged. In fact, in subsequent work, we were able to eliminate more than 25% of the source code–almost all of which was related to the now obsolete interface method that we replaced. More importantly, we were able to turn a code that was difficult to use and modify into a flexible package that was easy to use and extend. This has had a huge impact, which can not be understated, on the use of the SPaSM code to solve real problems.

6.2 Open-GL and Inventor

While SWIG was primarily designed to work with application codes, it can also be used to wrap large libraries. At the University of Utah, Peter-Pike Sloan used SWIG to wrap the entire contents of the Open-GL library into Tcl for use with an Open-GL Tk widget. The process of wrapping Open-GL with SWIG was as simple as the following :

- Make a copy of the `gl.h` header file.
- Clean it up by taking a few C preprocessor directives out of it. Fix a few typedefs.
- Insert the following code into the beginning

  %module opengl
  %{
  #include <GL/gl.h>
  %}

  ... Copy edited gl.h here ...

- Modify the Open-GL widget to call `Opengl_Init`.
- Recompile

The entire process required only about 10 minutes of work, but resulted in more than 500 constants and 360 functions being added to Tcl. Furthermore, this extension allows Open-GL commands to be issued directly from Tcl's interpreted environment as shown in this example (from the Open-GL manual [OpenGL]).

```
% ... open GL widget here ...
% glClearColor 0.0 0.0 0.0 0.0
% glClear $GL_COLOR_BUFFER_BIT
% glColor3f 1.0 1.0 1.0
% glOrtho -1.0 1.0 -1.0 1.0 -1.0 1.0
% glBegin $GL_POLYGON
%    glVertex2f -0.5 -0.5
%    glVertex2f -0.5 0.5
%    glVertex2f 0.5 0.5
%    glVertex2f 0.5 -0.5
% glEnd
% glFlush
```

Early work has also been performed on using SWIG to wrap portions of the Open-Inventor package [Invent]. This is a more ambitious project since Open-Inventor consists of a very large collection of header files and C++ classes. However, the SWIG library mechanism can be used effective in this case. For each Inventor header, we can create a sanitized SWIG interface file (removing alot of the clutter found in the header files). These interface files can then be organized to mirror the structure of the Inventor system. To build a module for Tcl, a user could simply specify which modules they wanted to use as follows :

```
%module invent
%{
... put headers here ...
%}

%include "Inventor/Xt/SoXt.i"
%include "Inventor/Xt/SoXtRenderArea.i"
%include "Inventor/nodes/SoCone.i"
%include "Inventor/nodes/SoDirectionalLight.i"
%include "Inventor/nodes/SoMaterial.i"
%include "Inventor/nodes/SoPerspectiveCamera.i"
%include "Inventor/nodes/SoSeparator.i"
```

While wrapping the Inventor library will require significantly more work than Open-GL, SWIG can be used effectively with such systems.

6.3 Building Interesting Applications by Cut and Paste

SWIG can be used to construct tools out of dissimiliar packages and libraries. In contrast to combining packages at an input level as one might do with with Expect, SWIG can be used to build applications at a function level [Expect][2]. For example, using a simple interface file, you can wrap the C API of MATLAB 4.2 [MATLAB]. This in turn lets you build a Tcl/Tk interface for controlling MATLAB programs. Separately, you could wrap a numerical

[2] SWIG can also be used to add extensions directly to `expect` and `expectk`.

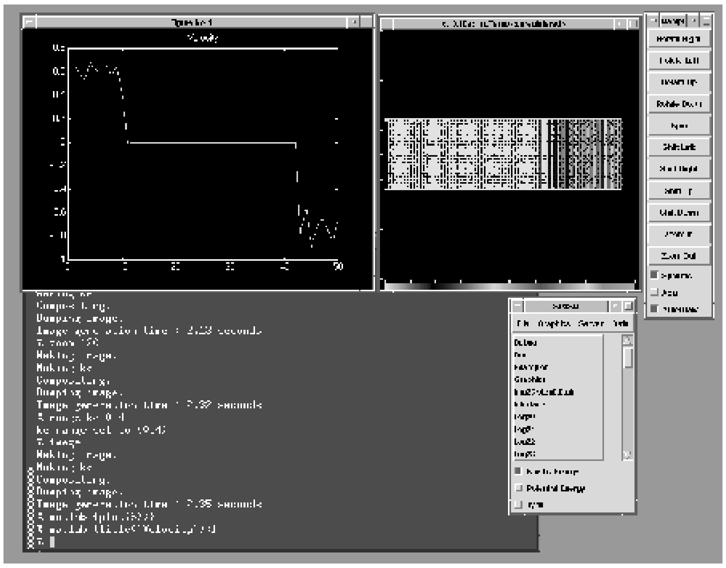

Figure 2: Simple Tcl/Tk Steering Application Built with SWIG

simulation code. Now, a few translation functions could be written and both packages combined into a single Tcl/Tk application. You have now built a simple "computational steering" system that allows you to run a simulation code, visualize data in MATLAB, and control the entire system with a Tcl/Tk based interface. Figure 2 shows a screen snapshot from such a simulation in which the SPaSM molecular dynamics code, a visualization module library, image server (using xv), and MATLAB have been combined. Under this system, the user can interactively set up and run simulations while watching the results evolve in real-time.

6.4 Language Independent Applications

Each scripting language has it's own strengths and weaknesses. Rather than trying to choose the language that is the best at everything (which is probably impossible), SWIG allows a user to use the best language for the job at hand. I'm always finding myself writing little utilities and programs to support my scientific computing work. Why should I be forced to use only one language for everything? With SWIG, I can put a Perl interface on a tool and switch over to Tcl/Tk at anytime by just changing a few Makefile options.

Since SWIG provides a language-independent interface specification, it is relatively easy to use SWIG generated modules in a variety of interesting applications. For example, the identical MATLAB module used in the last Tcl example could be imported as Perl5 module and combined with a Perl script to produce graphical displays from Web-server logs as shown in Figure 3.

Some have promoted the idea of encapsulating several languages into a higher level language as has been proposed with Guile [Lord]. This may be fine if one wants to write code that uses different languages all at once, but when I want to use only Tcl or Perl for an application, I almost always prefer using the real versions instead of an encapsulated variant. SWIG makes this possible without much effort.

6.5 Using Tcl as a debugger

Since SWIG can work with existing C code, it can be used quite effectively as a debugger. You can wrap C functions and interact with them from `tclsh` or `wish`. Unlike most traditional debuggers, you can create objects, buffers, arrays, and other things on the fly by putting appropriate definitions in the interface file. Since `tclsh` or `wish` provides a `main()` function, you can even rip small pieces out of a larger package without including that package's main program. Scripts can be written to test

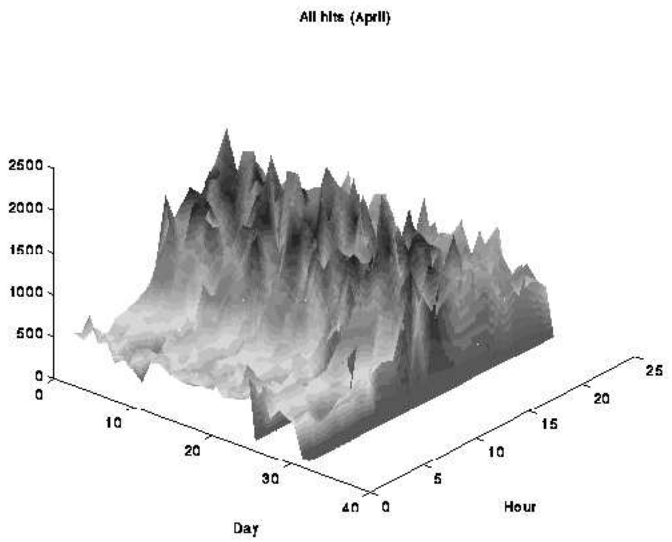

Figure 3: Web-server activity graph. Generated from a Perl script, but uses an unmodified SWIG generated module for integrating MATLAB with Tcl/Tk.

out different parts of your code as it is developed. When you are satisfied with the module, removing the interface is as easy as discarding the SWIG interface file. I have used SWIG as a debugger for developing graphics libraries, network servers, simulation codes, and a wide range of other projects–some of which didn't even use a scripting language when completed.

7 Limitations

SWIG represents a balance of flexibility and ease of use. I have always felt that the tool shouldn't be more complicated than the original problem. Therefore, SWIG certainly won't do everything. While I believe the pointer representation of complex datatypes works well in practice, some users have found this to be too general or confusing. SWIG's support for C++ is also limited by the fact that SWIG does not support operator overloading, multiple inheritance, templates, and a number of other more advanced C++ features. SWIG also lacks support for default or variable length function arguments, array notation, pointers to functions (unless hidden by a typedef) and an exception mechanism. Finally, SWIG does not support all of the features of Tcl such as associative arrays. These aren't an inherent part of the C language so it is difficult to generalize how they should be handled or generated. I prefer to keep the tool simple while leaving these issues up to the individual programmer.

Fortunately, it is almost always possible to work around many problems by writing special library functions or making modifications to the SWIG language modules to produce the desired effect. For example, when wrapping Open-GL, SWIG installed all of the Open-GL constants as Tcl global variables. Unfortunately, none of these variables were accessible inside Tcl procedures unless they were explicitly declared global. By making a modification to the Tcl language module, it was possible to put the GL constants into hash table and perform a hidden "lookup" inside all of the GL-related functions. Similarly, much of the functionality of SWIG can be placed into library files for use in other modules.

8 Conclusions

SWIG has been in use for approximately one year. At Los Alamos National Laboratory, its use with the SPaSM code has proven to be a remarkably simple, stable, and bug-free way to build and modify user interfaces. At the University of Utah, SWIG is being used in a variety of other applications where the users have quickly come to appreciate its ease of use and flexibility.

While SWIG may be inappropriate for certain applications, I feel that it opens up Tcl/Tk, Python, Perl, Guile, and other languages to users who would like to develop interesting user interfaces for their codes, but who don't want to worry about the low-level details or figuring out how to use a complicated tool.

Future directions for SWIG include support for other scripting languages, and a library of extensions. SWIG may also be extended to support packages such as incremental Tcl. Work is also underway to port SWIG to non-unix platforms.

9 Acknowledgments

This project would not have been possible without the support of a number of people. Peter Lomdahl, Shujia Zhou, Brad Holian, Tim Germann, and Niels Jensen at Los Alamos National Laboratory were the first users and were instrumental in testing out the first designs. Patrick Tullmann at the University of Utah suggested the idea of automatic documentation generation. John Schmidt, Kurtis Bleeker, Peter-Pike Sloan, and Steve Parker at the University of Utah tested out some of the newer versions and provided valuable feedback. John Buckman suggested many interesting improvements and has been instrumental the recent development of SWIG. Finally I'd like to thank Chris Johnson and the members of the Scientific Computing and Imaging group at the University of Utah for their continued support and for putting up with my wrapping every software package I could get my hands on. SWIG was developed in part under the auspices of the US Department of Energy, the National Science Foundation, and National Institutes of Health.

10 Availability

SWIG is free software and available via anonymous FTP at

`ftp.cs.utah.edu/pub/beazley/SWIG`

More information is also available on the SWIG homepage at

`http://www.cs.utah.edu/~beazley/SWIG`

References

[Beazley] D. M. Beazley and P. S. Lomdahl, *Message-Passing Multi-Cell Molecular Dynamics on the Connection Machine 5*, Parall. Comp. 20 (1994) p. 173-195.

[ET] Embedded Tk, `ftp://ftp.vnet.net/pub/users/drh/ET.html`

[Expect] Don Libes, *Exploring Expect*, O'Reilly & Associates, Inc. (1995).

[Heidrich] Wolfgang Heidrich and Philipp Slusallek, *Automatic Generation of Tcl Bindings for C and C++ Libraries.*, USENIX 3rd Annual Tcl/Tk Workshop (1995).

[Janssen] Bill Janssen, Mike Spreitzer, *ILU : Inter-Language Unification via Object Modules*, OOPSLA 94 Workshop on Multi-Language Object Models.

[Lomdahl] P. S. Lomdahl, P. Tamayo, N. Grønbech-Jensen, and D. M. Beazley, *Proc. of Supercomputing 93*, IEEE Computer Society (1993), p. 520-527.

[Lord] Thomas Lord, *An Anatomy of Guile, The Interface to Tcl/Tk*, Proceedings of the USENIX 3rd Annual Tcl/Tk Workshop (1995).

[MATLAB] *MATLAB External Interface Guide*, The Math Works, Inc. (1993).

[Ousterhout] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley Publishers (1994).

[Perl5] Perl5 Programmers reference, `http://www.metronet.com/perlinfo/doc`, (1996).

[vtk] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit*, Prentice Hall (1995).

[Invent] J. Wernecke,"The Inventor Mentor", Addison-Wesley Publishing (1994).

[Wetherall] D. Wetherall, C. J. Lindblad, "Extending Tcl for Dynamic Object-Oriented Programming", Proceedings of the USENIX 3rd Annual Tcl/Tk Workshop (1995).

[OpenGL] J. Neider, T. Davis, M. Woo, "OpenGL Programming Guide", Addison-Wesley Publishing (1993).