Scripting for EDA Tools: A Case Study

Pinhong Chen
Dept. of EECS
U. C. Berkeley
Berkeley, CA 94720, USA
pinhong@eecs.berkeley.edu

Desmond A. Kirkpatrick
Intel Corp.
Microprocessor Products Group
Hillsboro, OR 97124,
desmond.a.kirkpatrick@intel.com

Kurt Keutzer
Dept. of EECS
U. C. Berkeley
Berkeley, CA 94720, USA
keutzer@eecs.berkeley.edu

Abstract

How to integrate EDA tools to enable interoperability and ease of use has been a very time-consuming and complicated job. Conventionally, each tool comes with a unique and simple set of commands for interactive use such as Sis[12], Vis[8], and Magic[5], but it lacks full programming capability of a scripting language. Also, it discourages further exploration to the underlying system functionality. Not only the code is hard to reuse, but also rapid prototyping of a new algorithm is impossible. A new algorithm may still take years to develop, which has to start from scratch and struggles between various formats. In this paper, we study and address how to easily integrate those application program interface(API)'s into most popular scripting languages such as Tcl[10] or Perl[13]. This enables a full scripting or programming language capability into a tool, and most important of all, any tool can be interoperated over a uniform platform on an API level. Rapid prototyping of a new algorithm thus becomes much easier and faster. It also promotes software reuse. Many existing extension packages for the scripting languages can be therefore integrated such as Tk[10] for graphic user interface(GUI), and CPAN collection[4] for various Perl applications. From a standpoint of high software quality, this approach also provides a very good vehicle for comprehensive testing of each API in an EDA tool.

1. Introduction

Scripting languages such as Perl and Tcl[11] are well-suited for high level programming and system integration. The code required for a same task is usually less by a factor of 5X to 10X [11] compared with a system programming language such as C/C++ or Java. However, it is not efficient or optimal for performance oriented tasks, for which the traditional system programming languages can work very well. One approach naturally combines a scripting language at the top layer and uses dedicated and optimized algorithm engines from system programming languages for the underlying structures. This approach is very powerful, flexible, and easy for scripting or rapid prototyping of an application system. For EDA tools, as they are often characterized as an efficient core engine optimized for performance based on a system programming language, it often lacks the ability to integrate with the other existing systems, or has poor programming, scripting or customization capability. We propose an approach to fast linking EDA tools with scripting languages using an interface wrapper generator. A possible system configuration is shown in Fig.1, where a

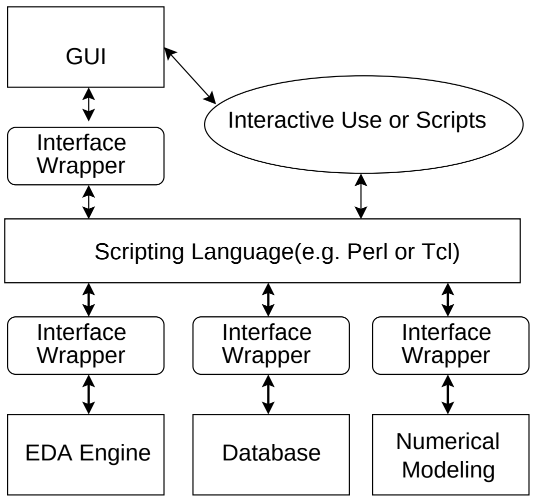

Figure 1. Integration of Tools by a Scripting Language

scripting language is used to integrate or ”glue” tools together, providing full programming capability to end-users. All of tools are dynamically loadable, that is to say, one can choose components based on the task need, and tool vendors can develop their own application system independently and hook it up later, or even create revisions without affecting the system integrity.

Scripting capability is usually important for major EDA tools. Commercial tools usually adopt a dialect of certain popular language such as Skill for Silicon Ensemble[1], Scheme for Apollo II[2], or Tcl for Design Compiler[3]. It provides API's access to automate a design task. However, many tools just adopt a simple and original interactive command shell, which is neither extensible nor flexible and not easy to program. Besides, a specific scripting language may be another stopping factor for a user to further explore the tool, due to a long learning curve for a new programming language.

Rapid prototyping and reuse of software components are a key to software productivity. Many high level algorithms are not even possible without the underlying database and supporting routines. It is thus desired that an EDA tool developer can implement a new algorithm efficiently by leveraging existing software components. Scripting languages can bridge the gap by providing a full programming environment and linking with existing tools from a higher level of language description without any compilation.

EDA tools have been lack of interoperability for a long time. There are several methods to communicate between two programs or working processes. Consider a delay calculator and a Verilog simulator as an example, which is shown in Fig. 2. The delay calculator will send delay data

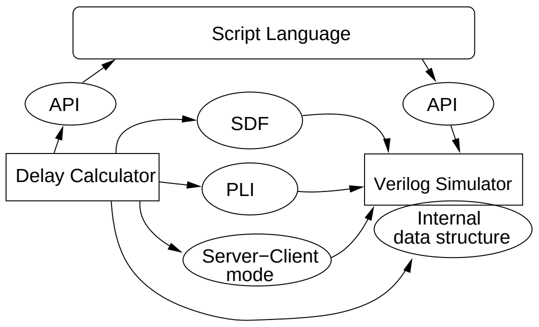

Figure 2. Two Communicating Processes

back to the Verilog simulator. There are several approaches to exchange data:

Text/Binary File The first approach uses some specific format to exchange data through normal OS files, for example, using standard delay format(SDF) to exchange delay data. This approach has serious drawbacks:

1. Extra dumping of data into the specific format and parsing of that format are required,
2. The input and output formats between two tools can be very possibly mismatched or misrepresented, i.e., format compatibility problem,
3. The input or output formats are thus fixed without any flexibility to change unless any further post processing is performed.

From a high productivity point of view, it is not desired to learn too many formal programming languages for a designer. It is not uncommon that a designer may have to understand at least a dozen of programming/scripting languages and formats to do a good design job just for using various EDA tools, and a dozen of supporting scripts may be required to transform formats between different tools.

Programming Language Interface The second approach uses a set of programming language interface(PLI)'s to communicate with each other. The delay calculator will provide service routines linked to a host, the Verilog simulator, through these PLIs. One has to be very careful about data types and usage details of these PLIs to make it work smoothly. It requires a separate linking pass to make it an executable simulator, resulting in a very time-consuming, non-extensible, and inflexible solution.

Client-Server Mode The third approach uses a client-server mode. For example, the delay calculator runs as a server waiting for the Verilog simulator to input information and feedbacks with delay data. It requires a detailed set of communication protocol to make this feasible. It is very time-consuming to implement and verify the complicated communication protocol and suffers from the overhead of extra communication and reliability problems.

Direct Access of Internal Data Structure The fourth approach uses internal data access. This is the most efficient one. However, due to data abstraction and code consistency, it is almost against all the software engineering principles. It is not only difficult to maintain, but also easy to crash a whole system.

API Access through a General Scripting Language We therefore propose that all the design tasks can be integrated onto a uniform platform to reduce the text/binary file exchange, and the end-users can access some of the APIs to do customization to fit their need using most popular scripting languages such as Perl or Tcl. The development time can be thus much reduced. Each component can be hooked up to that platform dynamically under the control of a script to complete a design task.

Comprehensive testing of a software routine is generally very difficult and time-consuming. The common testing approach is based on an outer input and output pair. It can not handle finer grain testing for any specific API. However, with the integrated APIs in the scripting language, a tool developer can design a set of very dedicated scripts to test each API and does not have to compile another testing programs to intervene with the production code. A series of regression tests for the API can thus be easily created to guarantee high software quality.

In general, integrating APIs into a popular scripting language is not very straight-forward. Frequently, there are lots of extra work required to make the interface self-consistent. We emphasize minimal extra coding to link a set of APIs into a scripting language, and provide some approaches to easing the integration work. SWIG[7] is designed to automatically generate the wrapper functions. It can reduce most of routine jobs into defining a simple configuration file and generate the required codes to bridge the gap.

In this paper, we study and address using SWIG and some techniques to interface and link the features or functions that an EDA tool may have to most popular script languages such as Tcl or Perl. These two scripting languages have been extensively used in the design community for a long time due to its powerful scripting capability, popularity, and extensibility. Both can process a simple text file very efficiently without the tedium to program a full parser and lexical analyzer. We will show how to integrate Tcl or Perl interfaces to a logic synthesis system Sis[12]. We will first introduce briefly the SWIG functionality. It is followed by various techniques used in our case study. We conclude by interface design consideration and propose for future work.

2. Review of an Interface Generator

Simplified Wrapper and Interface Generator(**SWIG**)[7] is a tool to generate necessary wrapper functions for C/C++ codes to interface with scripting languages. It can generate wrapper codes used to translate data types to fit the interface needs, including wrapping of input and output arguments for a C/C++ function. Moreover, it can generate data structure access routines to read or write C/C++ structures, variables or objects. The following is an simple example to show `sis_script.i` for the SWIG input configuration(interface) file.

```
%module sis_script
%{
#include "sis.h"
static network_t *current_network=NULL;
%}
network_t *current_network;
char *network_name(network_t *network);
void network_set_name(network_t *network,
         char *name);
%init %{
init_sis(0);
%}
```

One can run SWIG for linking with Tcl by

```
swig -tcl sis_script.i
```

A wrapper C file, `sis_script_wrap.c` will be generated, which includes a write function, `current_network_set` and a read function, `current_network_get` to access variable `current_network`. `network_name` will be wrapped up as a command in Tcl with one input variable, and it will return one string. Also, `network_set_name` will be wrapped up as a command in Tcl with two input variables. Tcl will initialize the code according to `%init` section. After invoking `tclsh`,

```
load ./sis_script.so
network_set_name  "a_new_name"
puts [network_name  [current_network_get]]
```

can be used to load the dynamic module of `sis_script` and start to work.

Note the same file can be used to generate Perl interface by

```
swig -perl5 sis_script.i
```

After linking with related libraries, one can invoke it by

```
use sis_script;
package sis_script;
network_set_name("a_new_name");
print network_name($current_network);
```

SWIG provides rich features to support interface and data type mapping functions plus many examples to show how to handle each case. It also supports linking with Python and Guile[7].

3. Interface Building for Tcl or Perl

In the rest of this paper, we will discuss the techniques required to link Sis APIs into Tcl or Perl. Most of the interface code can be generated by SWIG, but there are still some features in EDA tools requiring special care as discussed in the following.

3.1. Using Existing Command Dispatcher

For Sis, there is a default command dispatcher `com_dispatch` with an interface as

```
void com_dispatch(network_t* network,
         int argc, char* argv[]);
```

. We can leverage this command dispatcher interface by wrapping it with another routine for Tcl such as:

```
%{
static int _tcl_dispatch(ClientData clientData,
   Tcl_Interp *interp, int argc, char *argv[]){
   com_dispatch(&current_network, argc, argv);
   return TCL_OK;
}
%}
```

, and for Perl as:

```
%{
XS(_perl_dispatch) {
    char ** _arg0;
    dXSARGS ;
    int i;
    _arg0 = (char **)malloc((items+1)*
                     sizeof(char *));
    _arg0[0]=GvNAME(CvGV(cv));
    for (i = 1; i < items+1; i++) {
        _arg0[i] = (char *) SvPV(ST(i-1),na);
    }
    com_dispatch(&current_network,items+1,
                 _arg0);
    free(_arg0);
    XSRETURN(0);
}
%}
```

where `GvNAME(CvGV(cv))` can fetch the command Perl is just issuing.

We still have to register those commands in the command table when Tcl initializes:

```
%init %{
   avl_foreach_item(command_table, gen,
          AVL_FORWARD, &key, NIL(char *)){
      Tcl_CreateCommand(interp, key,
          _tcl_dispatch,(ClientData)NULL,
          (Tcl_CmdDeleteProc *) NULL);
   }
%}
```

or when Perl initializes:

```
%init %{
    char buf[4096];
    avl_foreach_item(command_table, gen,
            AVL_FORWARD, &key, NIL(char *)){
        sprintf(buf,"sis_script::%s",key);
        newXS(buf, _perl_dispatch, file);
    }
%}
```

With this translation, we can port the Sis command set into a Tcl dynamic extension library or a Perl package within 5 minuates of work. It is amazing that the original Sis scripts can run by this way in Tcl without any modification. For example, an original script `script.rugged` as shown below can run without any syntax modification:

```
sweep; eliminate -1
simplify -m nocomp
eliminate -1
sweep; eliminate 5
simplify -m nocomp
resub -a
fx
resub -a; sweep
eliminate -1; sweep
full_simplify -m nocomp
```

or in Perl with minor syntax differences:

```
use sis_script;
package sis_script;
sweep; eliminate -1;
simplify ’-m’,nocomp;
eliminate -1;
sweep; eliminate 5;
simplify ’-m’,nocomp;
resub ’-a’;
fx;
resub ’-a’; sweep;
eliminate -1; sweep;
full_simplify ’-m’,nocomp;
```

3.2. Information Extraction

The rich regular expression operations from Tcl or Perl can be used to extract run-time information for adaptively control based on the progress of optimization, for example, in Tcl:

```
proc get_map_result {} {
   set result [get_output\
       {map -s -n 1 -c 0}]
   regexp \
       {total neg slack:\s*\(([\d.-]+),([\d.-]+)}\
       $result dummy rise_slack fall_slack
   return [expr $rise_slack+$fall_slack]
}
set slack -10000000
while 1 {
   source script.algebraic
   set old_slack $slack
   set slack [get_map_result]
   puts  "Slack=$slack"
   if {$old_slack > $slack + 40} break
}
```

or in Perl:

```
sub get_map_result{
   my $result=get_output
      '&sis_script::map("-s","-n",1,"-c",0)';
   my ($rise_slack,$fall_slack)=
      ($result =~
      /total neg slack:\s*\(([\d.-]+),([\d.-]+)/o);
   return ($rise_slack+$fall_slack)/2;
}
$slack=-10000000;
while(1){
   require "script.algebraic.pl";
   $old_slack=$slack;
   $slack=get_map_result();
   print "Slack=$slack\n";
   if($old_slack > $slack + 40){ last;}
}
```

The function `get_map_result` is used to extract the total slack information. We implement a very useful output redirection routine `get_output` to catch the information from standard output and standard error. With the powerful regular expression operations from Perl or Tcl, one can extract any information from an API's output. This reduces the complexity to understand each detail inside the C/C++ data structure. Also, it is a crucial technique for black-box testing. For example, one can test a function `API_in_test` in Tcl as

```
set output [get_output API_in_test errlog]
if {regexp -nocase error $errlog} {
   puts "Testing of API_in_test fails!"
} else {
   puts "OK!"
}
```

3.3. Object Oriented Interface

Basically, SWIG provides features to assist this process. Note that even a pure C code can be translated into object-oriented style interface in scripting languages. Some macro definitions may be required to tune the data type and API's name. `%name` and `$rename` are supported in SWIG to change names.

Creation of an object interface requires some extra work in SWIG. First, we have to configure the declarations for an object in the SWIG configuration file. For example, `network` object is specified as:

```
typedef struct{ }network;
typedef network network_t;
%addmethods network{
     network(network_t *default=NULL);
     ~network();
     node_t* find_node(char *name);
     int num_pi();
}
```

Using this, SWIG will set `new_network` for the object constructor, `delete_network` for the object destructor, and adds a prefix `network_` for the other member functions in C/C++. Note that `typedef network network_t` can create an equivalent data type class in SWIG. For Tcl, one can thus create a network object as:

```
set network [network -this [current_network_get]]
```

, delete an network object by

```
rename $network {}
```

, or call a member function `find_node` by

```
$network find_node "input_node1"
```

For Perl, one can create a network object as:

```
$network=new network($current_network);
```

, delete an network object by

```
$network->DESTORY();
```

, or call a member function `find_node` by

```
$network->find_node "input_node1";
```

Note that one has to turn on `-shadow` in SWIG to generate the object oriented interface for Perl.

3.4. Translation of Foreach Style Construct

There are often pre-defined macros that will traverse some data structures to examine each object. This saves extra memory to build a pointer array. For example, in Tcl, one can use

```
foreach_node n $network {
    puts -nonewline \
       "Node [node_name $n] has fanin: "
    foreach_fanin p $n {
       puts -nonewline "[node_name $p] "
    }
    puts ""
}
```

to dump all the fanin of a node in `$network`.

The implementation of `foreach_node` command is very similar to the command `while` implementation in Tcl[10]. Although this implementation is quite complex, SWIG support a full C preprocessor capability. It can be used as a simple template generator to handle this type of macros.

For Perl, unfortunately, there is no corresponding syntax construct that can emulate this macro. One has to implement a function to return a list consisting of all the nodes in the network.

3.5. Callback Function

It is a common practice that one has to implement customized callback functions once an event has been triggered. The user has to register the callback functions before starting a process. However, since the callback function should be registered as a C/C++ function, we have to use a technique to pre-register it with a general callback function. It can be achieved by pre-registering a specific callback function in C and use `eval` command of Tcl to implement a callback function for Tcl, for example, we register a node allocation daemon in SWIG configuration file as

```
%init %{
    node_register_daemon(DAEMON_ALLOC,
        node_ALLOC_daemon);
%}
```

where `node_ALLOC_daemon` is implemented as:

```
static char *callback_function;
%{
static Tcl_Interp *stored_interp;
static char *callback_function=NULL;
void node_ALLOC_daemon(node_t *node){
    char cmd[1024];
    if(callback_function){
       SWIG_MakePtr(stored_interp->result,
           (void *)node,"_node_t_p");
       sprintf(cmd,"%s %s",callback_function,
           stored_interp->result);
       Tcl_Eval(stored_interp,cmd);
    }
}
%}
```

where `SWIG_MakePtr` is a wrapper function from SWIG to convert a pointer in C to Tcl representation, and `Tcl_Eval` is the C API for `eval` command in Tcl. The callback function can now be used in Tcl directly by:

```
set callback_function "processing_node_alloc"
proc processing_node_alloc {node} {
   # do something here for $node
}
```

There is a corresponding implementation in Perl as well[9].

3.6. Data Type Override in Scripting Language

For SWIG implementation, it encodes a pointer into a string with its hex address attached with datatype. However, in some situations, it may require a type transformation or data type casting. It can be done in Tcl as

```
regsub {^(_[0-9a-f]+)_.*$} $old_ptr\
    "\\1_$new_datatype" $new_ptr
```

or in Perl, due to SWIG's implementation, as

```
bless $old_ptr,$new_datatype;
```

3.7. Namespace Confliction

Since each module is developed independently, it is easy to have name conflictions when linking all modules together. Tcl has a command supporting `namespace` along with a SWIG switch `-namespace`. Perl supports `package` directive and `Exporter` class to avoid this problem[13].

Some C dynamic library may have name collisions as well. It can be solved by linking each package with `-Bsymbolic` switch or conservatively with `-Bstatic`.

3.8. Memory Management

In general, the user must be responsible for releasing unused memory. However, it is possible to set up a variable trace command in Tcl to automatically delete a local object:

```
trace variable $obj u delete_obj
proc delete_obj {name1 name2 op} {
    rename $name1 {}
}
```

For Perl, with `-shadow` switch, SWIG can emulate an object behavior naturally, the local variable is thus destroyed automatically by Perl's scoping mechanism.

3.9. Variable Trace

For common data types such as int, double, and string, SWIG can implement trace feature. That is to say, a Tcl variable will be created to follow the value changes in C/C++ code. However, for pointer types, SWIG will create read and write routines for each variable for Tcl. There are some tricky implementation techniques to use `trace` command to tie a Tcl variable and C/C++ variable together.

For Perl, SWIG uses a technique to tie the variable with associated access routines, so it is transparent to the user.

3.10. API Document Generation

SWIG supports document generation by extracting information from the interface configuration file. If the documentation is embedded in the C/C++ code as comments around an API, it can be extracted and organized as a html or Latex document.

4. Interface Design Consideration

The interface design is in general an art instead of allowing an exact analysis. It may require the interface to look like the original interface in C/C++ ; meanwhile, it must have some handy features in the scripting language such as list or hash data structures. Some helper functions in the scripting language may be needed to have an object interface.

4.1. Selective API Export

Although SWIG eases the job of interface creation, it may not be necessary to export every routine and data structure to a scripting language. However, it should provide complete and consistent functionality to access the core data structures. SWIG may also generate a large chunk of interface code to link with. It is also possible to reduce the object code size by stripping out the debug information.

4.2. Efficiency Issues

The efficiency issue is the major drawback for a scripting language. The wrapper and data types mapping function can create a lot of overhead for data transformation plus the memory management overhead to hold the data temporarily. However, as the technology for scripting languages improves, and speed and memory space increase for modern computers, it still allows interactive use, and more and more users are turning to use them, because of their ease of use and a rich set of application extensions.

This interface can speed up a new algorithm development a lot. The efficiency of the algorithm can be improved by rewriting scripting language subroutines into a C/C++ code without changing the calling convention on the higher level.

5. Conclusion and Future Work

We study how to integrate APIs of an EDA tool to the most popular and powerful scripting languages, providing an example to show the techniques for smooth and efficient integration.

Since all the features that an EDA tool might have for linking with a scripting language have been identified and solved above, we plan to create Tcl or Perl interfaces with this approach for more public modain EDA packages. They will be available at our web site[6] in the near future.

References

[1] *http://www.cadence.com*.
[2] *http://www.avanticorp.com*.
[3] *http://www.synopsys.com*.
[4] "Comprehensive Perl Archive Network". *http://www.cpan.org*.
[5] "Magic - A VLSI Layout System". *http://www.research.digital.com/wrl/projects/magic/magic.html*.
[6] "ScriptEDA". *http://www-cad.eecs.berkeley.edu/∼pinhong/scriptEDA*.
[7] "Simplified Wrapper and Interface Generator". *http://www.swig.org*.
[8] A. Aziz and et al. "VIS User's Manual". *http://www-cad.eecs.berkeley.edu/Respep/Research/vis/index.html*.
[9] D. M. Beazley, D. Fletcher, and D. Dumont. "Perl Extension Building with SWIG". *O'Reilly Perl Conference 2.0*, pages 17–20, Aug. 1998.
[10] J. K. Ousterhout. *"Tcl and the Tk Toolkit"*. Addison-Wesley, 1994.
[11] J. K. Ousterhout. "Scripting: Higher Level Programming for the 21st Century". *IEEE Computer magazine*, Mar. 1998.
[12] E. M. Sentovich and et al. "SIS: A System for Sequential Circuit Synthesis". *Electronics Research Laboratory Memo. No. ERL/UCB M92/41*, May 1992.
[13] L. Wall, T. Christiansen, and R. Schwartz. *"Programming Perl"*. O'Reilly and Assoicates, 1996.