GSC - A SystemC to Verilog translator

Samuel Shoji Fukujima Goto - RA:017335

Prof. Guido Araújo

19/06/2006

Instituto de Computação - IC/UNICAMP

email:samuel.goto@ic.unicamp.br

Abstract

This paper presents a comparison between two different hardware description languages - SystemC and Verilog -, and describes the development of a real life translator.

Contents

1 Introduction

1.1 What is verilog ?

In the past, when circuits were simple, hardware designers were satisfied with schematic models of digital circuits : it was an elegant and robust way to model a design. However, with the constant growth of project's size and complexity, describing circuits with wires and logical ports was getting impractical, and engineers started discussing how to describe hardware in a more convenient manner.

Hardware description languages were evolved from this schematic description, and for a while they were sufficient. Verilog - along with VHDL -, has been used as the industry standard for hardware descriptions for a while now. They have become famous because they are simple, have a similar syntax to C, and offers a great power over the project (just as much as schematic models) with the simplicity of a higher level of abstraction (know as register transfer level).

However, once again, digital circuits started getting too big and complex, and describing them in a hardware description language such as Verilog is becoming impractical for systems nowadays. Designers needed an even higher level of abstraction and better support for tools.

1.2 What is SystemC ?

SystemC is a C++ library that extends the C++ core to support hardware descriptions constructs. SystemC is used to model and describe hardware with all the benefits of the C++ infrastructure (compilers, editors, libraries, pre-processors, etc).

It supports and implements most of hardware data types - wires, signals, bits, registers, memory, etc - and all hardware language paradigms - parallel processing, asynchrony, etc.

Although SystemC is perfectly capable of describing hardware at the Register Transfer Level (RTL), it has been mostly used because it offers great support at a high level of

abstraction, the System Level.

1.3 What is SystemC RTL ?

The idea behind SystemC is that designers should have only one language to describe all levels of abstraction during the design flow. In the past, the verification team wrote their code in a high level language (such as C or Java) while the designers team were writing code in Verilog of VHDL.

The final description of a SystemC module is a Register Transfer Level model of the design, witch should be as detailed as it would be in any other language.

The Register Transfer Level of a language is a subset of this language, that can be synthesized by a synthesis tool.

1.4 Filling the gap

Verilog and VHDL has been know as the standard for hardware description languages in the industry. Although SystemC is capable of describing RTL modules, it will take a while for the industry to accept it as a standard.

What is being presented on this paper is an attempt to produce a real life translator of SystemC RTL to Verilog RTL, a verification methodology and a discussion on the results.

2 The Problem

To write a compiler (or a translator) one must handle three basic parts : the front end (parsing), elaboration (verification and optimization), and the back end (code emission). In this chapter I will discuss these three basic steps, and show how I made some of the design decisions.

2.1 Parsing C++ : Keystone

I have seriously considered writing a c++ parser from scratch, but I have quickly learned a lesson : parsing c++ is not an easy task.

As stated before, parsing c++ could be a tedious and laborious work. It is not a difficult work per se, but taking care of each c++ ambiguity and syntactical use case would take a lot of time.

However, as strange as it may seem, trying to use an existent one isn't as easy as writing one from scratch. Fortunately, after a long search, there was one good solution : keystone [1].

Keystone is a c++ front end built to be a c++ front end : nothing more. There is no need to extract the c++ parser from a project or write a new one from scratch; keystone parses c++ source code and returns an AST representation in a graphviz dot format.

2.2 Abstract Syntax Tree : Graphviz

A good implementation decision on an AST representation is definitely time well spent. So I took a while and found a good graph modeling package called graphviz. It supports most of the graphs operators and transforms and it is very stable. Graphviz also generates graphical images of a graph, witch makes development easier [2].

2.3 Previous Work

Before gsc, there were other attempts to produce a SystemC to Verilog translator. Unfortunately, some of the most successful efforts are highly cost commercial products.

[1] http://www.cs.clemson.edu/ malloy/projects/keystone/doc.html

[2] http://www.graphviz.org/

Translators like Synopsy's dc shell and Forte's cynthVLG are great at their job, but not everyone can have unlimited access to them (by unlimited I mean "forever and at a reasonable cost").

There are, however, some attempts to produce an open source translator. Two of them are worth noting : sc2v and tabajara. The first is a opencores.org attempt, produced by an European group. The second is a Brazilian attempt. They are both very immature, and they are not ready to use in a normal design flow.

Their immaturity resides on one small but crucial design decision : they both tried to write a SystemC grammar to parse models and preprocessor directives. In effect, this design decision ties the code to a very strict c++ subset, witch makes general programming impractical : you will have to write each construct exactly how the front end wants you to.

Although I have tried myself to write a c++ parser, I have quickly noticed that writing a decent one is just not feasible in a short time (it would actually be a project on its own). Not using a well established preprocessor like cpp didn't seem like a good design decision either.

3 Code Emission

After SystemC RTL code is parsed and represented in an AST, each AST node is visited to extract information about the module. Information like module interface, signals and variable declarations, methods and sensitivity lists and modules hierarchy is extracted with a simple AST walk.

Each SystemC construct is then translated to a Verilog correspondent. Table 1 shows an example of the basic module structure.

3.1 sc_module

Table 1: Basic syntax of a module

SystemC	Verilog

`#include <systemc.h>`
 	
`SC_MODULE( half_adder ){`
`  sc_in< bool > a,b;`
`  sc_out< bool > sum, carry;`
 	
`  void prc_half_adder(){`
`  sum = ( a ^ b );`
`  carry = ( a & b );`
`  }`
 	
`  SC_CTOR( half_adder ){`
`  SC_METHOD( prc_half_adder );`
`  sensitive << a << b;`
`  }`
`}`
	
	`module half_adder(a, b, sum, carry);`
	`  input a;`
	`  input b;`
	`  output sum;`
	`  output carry;`
	`  reg sum;`
	`  reg carry;`
	
	`  always @( a or b )`
	`  begin : prc_half_adder`
	`  sum <= ( a ^ b );`
	`  carry <= ( a & b );`
	`  end`
	`end module;`

The basic container in SystemC is called a SC_MODULE(name). SC_MODULE(name) is actually a macro that expands into a c++ class declaration. Since we have total control over the pre processor (where macros are expanded), I have created a file called systemc.h with several systemc macros that expands as I find convenient. The SC_MODULE(name macro) actually expands as the following :

```
#define SC_MODULE( module ) class module
```

This is useful since as you walk on the AST, it is good to have pointers for keywords like SC_MODULE, SC_METHOD, etc. So basically, when we walk the AST and find a node like

```
class half_adder{
  // AST child nodes
}
```

we consider this a module declaration, witch will be further translated to

```
module half_adder( );
  // AST child nodes
end module;
```

3.2 sc_types

Each SystemC data type must have a Verilog correspondent(See Table 2).

Table 2) shows simple data width and sign conversions. Since Verilog isn't a strongly typed language (like VHDL), it greatly facilitates the translation job.

3.3 sc_input, sc_output, sc_inout

Input and output ports are extracted from the AST and translated into verilog. For example, port declarations like

Table 2: SystemC to Verilog data types mapping

SystemC data types	Verilog data types
sc_logic var	reg var
sc_bool var	reg var
int var	reg signed[31 : 0] var
sc_int< n > var	reg signed[n-1 : 0] var
sc_uint< n > var	reg [n-1 : 0] var
sc_bigint< n > var	reg signed[n-1 : 0] var
sc_biguint< n > var	reg [n-1 : 0] var

```
SC_MODULE( counter ){
        sc_in < bool > clk;
        sc_out < int > value;

        //statements
}
```

Is translated into :

```
module counter( clk, value );
        input clk;
        output [ 31 : 0 ] value;
        reg [ 31 : 0 ] value;

        -- statements
end module;
```

3.4 sc_signals vs variables

An important - and difficult - translation decision must be taken when translating signals and variables to Verilog. According to the SystemC language specification, signals behave much like Verilog registers assignment : they are non blocking, and can be accessed on different parallel threads (Table 3). Variables, however, behave much like wires assignments, since they are blocking and can only be accessed in a given scope (Table 4).

They are both declared as regs, but the difference lies on the kind of assignment they take during code execution : <= or :=.

The exception is when signals are used to connect sub modules in a hierarchy. In this case, signals should be considered and declared wires (Table 5).

Table 3: sc_signal to reg translation example

SystemC	Verilog

SC_MODULE(example){
sc_in< **bool** > clk;
sc_signal< **bool** > a;
 	
void prc_example(){
a = 1;
}
 	
}
	module example(clk);
	input clk;
	reg a;
	
	always @(clk)
	begin : prc_example
	a <= 1;
	end
	
	end module;

3.5 Hierarchy

There are many ways to describe hierarchy in SystemC (mainly because SystemC modules are basically classes, so one can instantiate a class in any C++ valid statement).

Table 4: variable to reg translation example

SystemC	Verilog

SC_MODULE(example){
sc_in < **bool** > clk;
bool a;
 	
void prc_example(){
a = 1;
}
 	
}
	module example(clk);
	input clk;
	reg a;
	
	always @(clk)
	begin : prc_example
	a := 1;
	end
	end module;

A full featured translator should be able to translated at least the conventional instantiation construct. Consider , for instance, the example on Table 5.

3.6 sc_method

Methods are the basic RTL execution model of SystemC. It is much like any other Verilog thread : it is a sequence of statements and expressions.

Each SystemC method has a sensitivity list associated, and is triggered every time an event occurs in it. You can have positive edge triggers, as well as negative edge triggers.

SystemC sensitivity list of methods prototypes are translated as in Table 6

Inside a SystemC class declaration, modules can have internal functions and procedures. One can identify if a method is a thread or a normal function if there is a SC_METHOD(func) statement inside the class constructor. Threads and functions are translated differently, as Table 6 shows.

Table 5: sc_signal to wire translation due to hierarchy interconnection

SystemC

```
SC_MODULE( full_adder ){
    sc_in<bool> a,b,cin;
    sc_out<bool> sum,cout;

    sc_signal<bool> c1,s1,c2;

    half_adder *ha1_ptr,*ha2_ptr;
    void prc_or()
      {
        cout = c1 | c2;
      }
    SC_CTOR( full_adder )
      {
      ha1_ptr = new half_adder();
      ha1_ptr->a( a );
      ha1_ptr->b( b );
      ha1_ptr->sum( s1 );
      ha1_ptr->carry( c1 );
      ha2_ptr = new half_adder();
      ha2_ptr->a( s1 );
      ha2_ptr->b( cin );
      ha2_ptr->sum( sum );
      ha2_ptr->carry( c2 );

      SC_METHOD( prc_or );
      sensitive << c1 << c2;
      }
};
```

Verilog

```
module full_adder( a, b, cin,
                   sum, cout );
    input     a;
    input     b;
    input     cin;
    output    sum;
    output    cout;
    wire      sum;
    reg       cout;
    wire      c1;
    wire      s1;
    wire      c2;
    always @( c1 or c2 )
    begin : prc_or
        cout  <= c1 | c2;
    end
    half_adder ha1_ptr(
            .a(a),
            .b(b),
            .sum(s1),
            .carry(c1)
    );
    half_adder ha2_ptr(
            .a(s1),
            .b(cin),
            .sum(sum),
            .carry(c2)
     );
end module
```

Table 6: SystemC to Verilog methods and functions translation

SystemC	Verilog
void prc_half_adder(){} SC_METHOD(prc_half_adder); sensitive << a << b << clk.pos();	**always** @(a **or** b **or** **posedge**(clk)) **begin** : prc_half_adder **end**

sc_uint<2>
func(sc_int<2> a, sc_int<2> b) {
  if(a–b <0)
  return(b–a);
  return(a–b);
}
	function[1:0] func;
	input [1:0] a;
	input [1:0] b;
	begin
	  if(a – b < 0) **begin**
	  func = b – a;
	  end
	  func = a – b;
	end
	endfunction

3.7 Statements and Expressions

Each c++ statement and expression inside a method in the AST is recursively translated with a verilog correspondent. C++ statements and expressions have similar verilog constructs . See Table 7.

4 Verification

It turns out that verifying a translation model is as difficult as building one. This is not a new thing for hardware verification teams, but it may not seem reasonable at first. The first problem arises if you don't have (or actually can't have) a testbench for each module you translate. In fact, there is no way to completely know if your translation describes your SystemC description perfectly. Having a translation tool (hypothetically fully verified) doesn't help you either : you may have two different translation that perfectly describes one circuit (use of #define pragmas instead of constants, for instance).

However, you can have a good idea of your translation efficiency if you cover some basic structure of a system description (if it helps, remember that this is an engineering paper, not a mathematician's).

In general, a translation should do exactly as it is told : a simple translation. It shouldn't fix a designer mistake or make assumptions. It must, however, make an interpretation of each piece of code and translate it. Is this chapter I present the verification methodology adopted by gsc and justify in witch case it is sufficient.

4.1 Syntax

gsc assumes that the SystemC model is syntactically and semantically right. This means that in a normal design flow, a designer would first create and validate a SystemC model

Table 7: SystemC to Verilog statements and expressions mapping

SystemC Statements	Verilog Statements
state = write_s;	state <= write_s;
state.write(write_s);	state <= write_s;
var = state.read();	var <= state;
var = a.range(1, 2);	var = a(1 : 2);
a {+,−,/,∗,&,|,&&,||} a	a {+,−,/,∗,&,|,&&,||} a
{−,!,˜} a	{−,!,˜} a

if (a){
}
else if (b) {
}
else{
}
	if a then
	end
	else begin
	if b then
	else
	end
	end

switch(a){
case 0 : { **break**; }
case 1 : { **break**; }
default : { **break**; }
}
	case (a)
	0: **begin end**
	1: **begin end**
	default : **begin end**
	endcase

for(i = 0 ; i < 3 ; i++)	**for**(i = 0 ; i < 3 ; i = i + 1)
i++	i = i + 1;
if (i++)	tmp = i + 1; if (tmp)
a = b ? c : d;	a <= b ? c : d;
10,0x10	10,'h10
a = memory[address];	a <= memory[address];
return a + b;	func_name = a + b;

before translating it to Verilog (witch implies compiling with an external c++ compiler like gcc or g++. This compilation and execution process should guarantee that the model is syntactically and semantically right, in terms of C++ statements and expressions. For the same reason, it is also safe to assume that all preprocessor directives (#defines, #ifdef, macros) and includes are resolved. This is perfectly reasonable to assume and greatly facilitates the translation.

Considering this assumption, gsc doesn't need to check and elaborate much over type checking, undefined references, undeclared variables and most of the common problem a normal compiler would. If gsc translates each c++ type and construct with a idempotent one, it may safely assume its validation.

Therefore, the first verification step it takes is through a syntactical analysis and verification made by a third party tool (to guarantee an external interpretation of the verilog language). gsc translates the SystemC code and feeds its translation to the iverilog [3] verilog compiler, witch makes most of the static type checks and references.

4.2 Semantics

A translator must emit syntactically valid code. This is easy to check. It must, however, generate a true and valid code that perfectly represents the host code in the target language.

It turns out that this is very difficult to do, and a translation tool is not enough to guarantee that the translation is perfect in all cases. The engineering solution to this problem is to build a subset of pairs (a : possible input, b : correspondent valid output) and feed this subset to the model.

There are EDA tools that supports the simulation of two different hardware languages (like Verilog plus SystemC, for example), called co simulation tools. gsc uses one of this tools, called Modelsim, to guarantee that for a given input vector, the translation behaves

[3] iverilog, also know as icarus verilog, is a verilog simulator generator, witch, in this case, is being used just as a verilog syntax verification tool

just like the original model.

4.3 Synthesis

A part from being semantically right, a translation tool should fit a synthesis tool as much as it can, after all, that is why it is being used : synthesis. Therefore, a synthesis tool called Altera Quartus 5.0 has been used on each test to check if it understands its translations.

4.4 Comercial Reference

Having all those steps resolved is enougth to have a good translator, but it can't offer any guarantee on its translations. Acctually, it is very hard (perhaps impossible) to have a fully verified translation tool.

However, having well stabilished and well tested tools in the hardware industry, that offers exactly what we are trying to develop, helps to have an idea of an 'accepted' translation result. Therefore, comparing results from third part tools results (usualy comercial's) of the same model should be enought to cover most of SystemC use cases.

5 Results

In this chapter I will present the results from several model's translations. Choosing the right test set was a big part of my work, and it is worth noting witch criterea were used and how they were analysed.

5.1 Testbench

Initially, before gsc tries to solve complex models, it should be able to translate simple and self contained examples of SystemC use cases. Things like adders, fsm [4], basic structures, hiearchy, functions, etc are included in the 'basic models' testbench. See results on Table 8.

After that, since gsc was born to cover, at least, as much as its predecessors, tests included in previous works (sc2v and tabajara) were used. All tests dispatched in the oficial distribution of sc2v and tabajara were used. See results on Table 9 and Table 10.

Finally, real world models - a mp3 and mpeg decoder - were used to guarantee that gsc could handle big and complex designs, written by external designers. See results on Table 11 and Table 12.

Comparisons were taken using Forte's CynthVLG tool.

5.2 Coverage

In this section I will discuss my experience with each tool, and point out their attributes (Table 13).

6 Next steps

gsc has proved to be a notable alternative to comercial tools, and has definitly proved its coverage superiority over other open source tools. gsc has been developed under solid basis (a full featured front end, its preprocessor and a well stabilished abstract syntax tree representation) and could be extended to support other features.

[4] finite state machines

Table 8: basic translations

model	description	syntax	semantics	synthesis	comparison
switch.cpp	switch() construct translation	pass	pass	pass	pass
func.cpp	function translation	pass	pass	pass	pass
half_adder.cpp	half adder	pass	pass	pass	pass
full_adder.cpp	full adder (hiearchy)	pass	pass	pass	pass
fsm.cpp	finite state machine	pass	pass	pass	pass
sequencia101.cpp	101 recognition of a binary stream	pass	pass	pass	pass
processor.cpp	a very simple processor	failed[a]	pass	pass	pass
cast.cpp	data type casts	failed	failed	failed	failed
init.cpp	data intialization	pass	pass	pass	pass
pp.cpp	if(i++) use case	failed	failed	failed	failed

[a]processor syntax verification failed because iverilog can't handle verilog 2001 witch does supports multidimensional arrays (iverilog covers only 1995 Verilog). However, this is a perfectly valid verilog featured and can be safely used

Table 9: sc2v testbench translations

model	description	syntax	semantics	synthesis	comparison
delay_line.cpp	not avaiable	pass	pass	pass	pass
half_adder.cpp	not avaiable	pass	pass	pass	pass
md5.cpp	not avaiable	pass	pass	pass	pass
sc_ex1.cpp	not avaiable	pass	pass	pass	pass
stmach_k.cpp	not avaiable	pass	pass	pass	pass
subbytes.cpp	not avaiable	pass	pass	pass	pass

Table 10: tabajara testbench translations

model	description	syntax	semantics	synthesis	comparison
dcdct.cpp	not avaiable	pass	pass	pass	pass
si.h	not avaiable	pass	pass	pass	pass

Table 11: mp3 testbench translations

model	description	syntax	semantics	synthesis	comparison
dct	not avaiable	pass	pass	pass	pass
imdct	not avaiable	pass	pass	pass	pass
overlap	not avaiable	pass	pass	pass	pass
reorder	not avaiable	pass	pass	pass	pass
imdctwindow	not avaiable	pass	pass	pass	pass
avalon	not avaiable	pass	pass	pass	pass
window	not avaiable	pass	pass	pass	pass
crc	not avaiable	pass	pass	pass	pass

Table 12: mpeg testbench translations

model	description	syntax	semantics	synthesis	comparison
bitstream	not avaiable	pass	not avaiable	pass	not avaiable
cbp	not avaiable	pass	pass	pass	pass
dcdct	not avaiable	pass	pass	pass	pass
piacdc	not avaiable	pass	not avaiable	pass	not avaiable
qi	not avaiable	pass	not avaiable	pass	not avaiable
rgb	not avaiable	pass	not avaiable	pass	not avaiable
si	not avaiable	pass	not avaiable	pass	not avaiable
sum	not avaiable	pass	not avaiable	pass	not avaiable
mem	not avaiable	pass	pass	pass	pass

Table 13: Existing alternatives comparison

Features	Synopsys	Forte	sc2v	tabajara	gsc
free	no	no	yes	yes	yes
open source	no	no	yes	yes	yes
front end	full featured	full featured	c++ subset[a]	c++ subset[b]	full featured[c]
type casting support	no	yes	no	no	partial
functions support	no	partial	yes	yes	yes
inout ports	yes	no	yes	yes	yes
basic testbench coverage	full	full	partial	partial	full
sc2v testbench coverage	full	full	full	failed	full
tabajara testbench coverage	full	full	failed	full	full
mp3 testbench coverage	full	full	failed	failed	full
mpeg testbench coverage	full	full	failed	full	partial

[a] hand written bison grammar

[b] hand written bison grammar

[c] keystone parser and cpp preprocessor

A part from supporting the remaining mandatory features (specially better type casts handling, and the if (i++) problem), there are other desired features that may find its path trought gsc roadmap.

6.1 High Level Synthesis

High Level Synthesis is having a lot of attention in the Computer Science field [4]. It basicaly tries to generate a hardware description from any high level description, typically a C program. It means that you could be able to describe hardware much like you describe any other program (witch is basicaly a description of what to do, an algorithim).

High Level Synthesis has been studied using a Control Data Flow Graph [1], with some kind of Instruction Scheduling Algorithim [2] and Resource Allocation Algorithim[3], subjected to a finite resource, timing and power constraint.

Future works on gsc will probably include some form of scratch in this particular subject.

6.2 Applications : Synthesis of ArchC Processors

A part from high level synthesis, gsc, as it is today, could be used for several applications : it is a generic RTL translation tool.

SystemC is getting its respect from its ability to handle big and complex projects, specially because of its System Level support. ArchC [5] is an ADL[6] for description and development of processors [5].

Altought ArchC current generates simulation-only models of processors (for runtime performance purposes), it is perfectly capable of generating RTL models of processors architecture (as long as the designer also writes RTL description of instructions too. Hand written scheduled pipelined instructions are a good example of RTL instruction). Since

[5] http://www.archc.org

[6] Architecture Description Language

gsc basicaly translates RTL models, it would be a nice feature to include gsc in the ArchC roadmap development and have synthesis of real processors take part of the ArchC design flow.

References

[1] Namballa, R.;Ranganathan, N.; Ejnioui, A.; Control and Data Flow Graph Extraction for High-Level Synthesis

[2] Memiki, S.O. ;Fallah, F; Accelerated SAT-based scheduling of Control/Data Flow Graphs

[3] Zhong, L..;Luo J.; Fei, Y.; Jha, N.; Register Binding based Power Management for High-level Synthesis of Control-Flow Intensive Behaviours

[4] Arvind;Rosenband, D. L.; Nikhil, R.S; Dave, N.; High Level Synthesis : An Essential Ingredient for designing Complex ASICs

[5] Azeved, R.;Rigo, S.;Bartholomeu, M.;Araújo, G.;Araújo, C.; Barros, E.; The ArchC Architecture Description Language