An Automatic Testbench Generation Tool for a SystemC Functional Verification Methodology

Karina R. G. da Silva
Universidade Federal de Campina Grande
Aprígio Veloso Avenue, 882, Bodocongó
Campina Grande - PB
Brasil
Karinarocha@dee.ufcg.edu.br

Elmar U. K. Melcher
Universidade Federal de Campina Grande
Aprígio Veloso Avenue, 882, Bodocongó
Campina Grande - PB
Brasil
elmar@dsc.ufcg.edu.br

Guido Araujo
Universidade Estadual de Campinas
Albert Einstein Avenue, 1251
Campinas - SP
Brasil
guido@ic.unicamp.br

ABSTRACT

The advent of new 90nm/130nm VLSI technology and SoC design methodologies, has brought an explosive growth in the complexity of modern electronic circuits. As a result, functional verification has become the major bottleneck in any design flow. New methods are required that allow for easier, quicker and more reusable verification. In this paper we propose an automatic verification methodology approach that enables fast, transaction-level, coverage-driven, self-checking and random-constraint functional verification. Our approach uses the SystemC Verification Library (SCV), to synthesize a tool capable of automatically generating testbench templates. A case study from a real MP3 design is used to show the effectiveness of our approach.

Categories and Subject Descriptors: B.7.3 [Integrated Circuits]: Reliability and Testing

General Terms: Verification.

Keywords: SystemC, SCV, VeriSC, Brazilip, tool.

1. INTRODUCTION

The most difficult challenge in the design of any system is to make sure that the final implementation is free of implementation flaws [2]. The goal of functional verification is to verify all functionalities of the design and to assure that it behaves according to the specification. To do this, one must create the design environment, by means of a testbench, that is capable of generating input data, while monitoring the design output against the output of a given reference model. Verification can consume over 70% of the overall design effort [1], and thus, tools that can quickly create efficient testbenchs are in great demand.

Some well established hardware description languages, like VHDL and VERILOG, are sometimes also used to do design verification. Although they have some useful hardware programming constructs, they lack major functional verification capabilities like constrained randomization, functional coverage and transaction recording. On the other hand, programming languages like C, C++, and Java, allow high-level abstraction constructs, but do not have the mechanisms to account for parallelism and timing which are required for hardware description. In order to close this gap, some specialized verification languages have been created like Verisity, OpenVera and the SystemC Verification Library (SCV).

Historically, several methodologies have been used for functional verification[7][8], but they lack generality and ease of use. There are currently (circa April 2004) no tools that are complete and generic enough to solve this problem [1]. For this reason, a lot of effort has been concentrated on the research of this problem.

The methodology proposed in this paper creates a SystemC based object oriented environment to perform verification. Many reuse methodologies are based on object-orientation[6]. In [4] the authors describe an approach that uses constraint solving to generate input vectors through a finite state machine. The machine produces all possible inputs to a specific Device Under Verification (DUV). In [5] the authors propose a methodology and a tool to do transaction-based functional coverage. Most tools cover some aspect of verification or are specific to some kind of DUV.

In this paper we propose a new methodology, that automatically creates a DUV-specific template testbench. The methodology intends to be generic and covers all kinds of synchronous DUV. Our methodology is implemented in VeriSC, a tool that performs automatic testbench generation. VeriSC uses SystemC and the SystemC Verification Library(SCV) to create a random-constraint, coverage-driven, self-checking and transaction-based testbench template. A case study from a real MP3 design is used to show the effectiveness of our approach.

The remainder of this paper is organized as follows. In section 2 we explain the proposed methodology. Section 3 describes the implementation from VeriSC. In section 4 we show a functional verification application example (MP3 design), followed by the conclusion in 5.

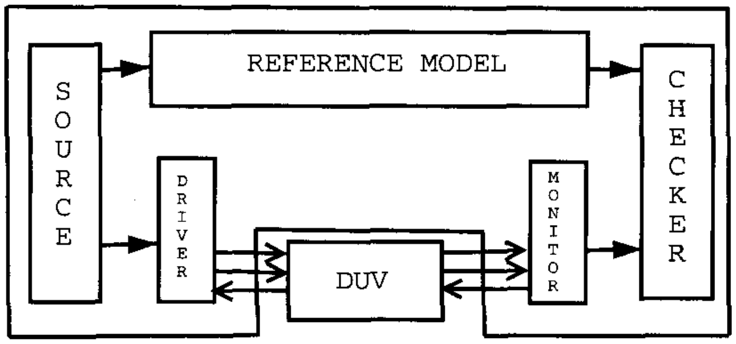


Figure 1: Testbench to Functional Verification

2. THE METHODOLOGY

An important problem in design verification is the need to adapt the testbench to the DUV. With our methodology, the verification engineer can use the testbench generation tool to make such adaptation. Other interesting aspect of this methodology is that the Reference Model (RM) can be written in virtually any high-level language, making it simple and easier to maintain. However, the method proposed can not be applied to reference models written in a non-executable description (e.g. natural language).

Our methodology proposes to create a testbench composed of source, driver, monitor, reference model and checker modules as shown in Figure1. Input data is fed into the DUV and the RM, and the outputs of both are collected to see if they are equivalent. The purpose of each module inside the functional verification methodology is explained next subsections.

We have been created a tool that automatize the testbench implementation. This tool follows the methodology showed in this section.

2.1 VeriSC

VeriSC implements the methodology proposed and automatically creates a template testbench according to the particular characteristics of the DUV. It creates all testbench modules: source, driver, monitor, reference model, checker and all FIFOs that connect these modules. The tool is also responsible for connecting the specific DUV. The template is created automatically by analyzing the DUV input and output ports and the SystemC descriptions of the transaction level structures. The transaction level structures must contain all information about the semantics of the relevant data that are communicating with the DUV.

VeriSC offers enhanced productivity to verification engineers by reducing the design time spent in creating testbenches. The resulting testbench templates are compact, easy to understand and guaranteed to compile and simulate without run time errors or hang ups. The next subsection describes VeriSC implementation details.

2.2 Automatic Template Generation

The VeriSC tool generates all templates of the testbench's modules according to the DUV. Signal handshake, functional coverage metrics and input value distributions must be implemented by the verification engineer.

For the sake of clarity a simple adder will be used as an example. The generated testbench, with the DUV and the RM is shown in Figure 2.

```
// interface in input
struct add_input
{ int a;
  int b;
};
inline ostream& operator << (ostream& os, const add_input& arg){
   os << "a=" << arg.a<< "b=" << arg.b;
   return os;
}

// interface out output
struct add_output
{ int s;
  inline bool operator == (const add_output& arg) const {
     return( ( s== arg.s) );
  }
};
inline ostream& operator << (ostream& os, const add_output& arg){
   os << "s=" << arg.s;
   return os;
}
```

Figure 3: Transaction-level structure

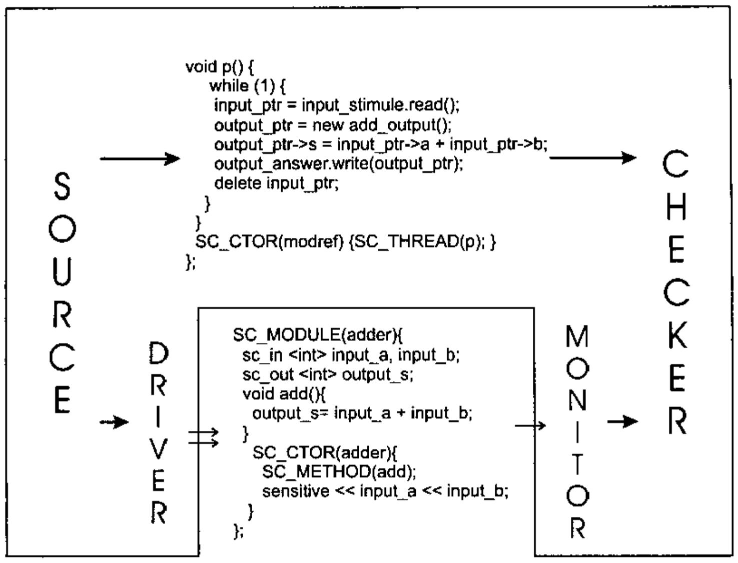
A parsing phase produces all the necessary information to construct the driver and checker ports and the FIFOs and to connect the DUV to the testbench. This phase also reads the specific transaction level structures given by the verification engineer. The drivers and monitors are generated straight from the transaction description file. The tool generates one driver for each input interface and one monitor for each output interface. Figure 3 shows our example transaction level structure.

The source module is created as an SCV class that inputs transaction level data into the DUV, see example in Figure 4. VeriSC creates one input class to each input interface that communicates with the DUV through FIFOs.

The Driver is responsible for transforming transaction-level data to handshake signals and pass for the DUV. There is one driver for each input interface of the DUV.

The implementation of the specific handshaking protocol for the DUV has to be done by the verification engineer using behavioral SystemC. The driver records each transaction for visualization.

The monitor is responsible for receiving the DUV's signals and transforming them into transaction level data. For the



```
void p() {
   while (1) {
    input_ptr = input_stimule.read();
    output_ptr = new add_output();
    output_ptr->s = input_ptr->a + input_ptr->b;
    output_answer.write(output_ptr);
    delete input_ptr;
  }
 }
 SC_CTOR(modref) {SC_THREAD(p); }
};
```

```
SC_MODULE(adder){
  sc_in <int> input_a, input_b;
  sc_out <int> output_s;
  void add(){
   output_s= input_a + input_b;
  }
   SC_CTOR(adder){
    SC_METHOD(add);
    sensitive << input_a << input_b;
  }
};
```

Figure 2: Adder's testbench generated

```
class input_constraint_class: public scv_constraint_base {
 scv_bag<int> a_distrib;
 scv_bag<int> b_distrib;
 public:
   scv_smart_ptr<add_input> add_input_sptr;
   SCV_CONSTRAINT_CTOR(input_constraint_class) {
      a_distrib.push(0, 50);
      a_distrib.push(1, 50);
      b_distrib.push(0, 50);
      b_distrib.push(1, 50);
      add_input_sptr->a.set_mode(a_distrib);
      add_input_sptr->b.set_mode(b_distrib);
   }
};
```

Figure 4: generated Source module

structure shown in Figure 3, with only one output interface, one monitor was generated. The monitor puts the data into a FIFO and passes them to the checker module.

The Checker module is responsible for functional coverage. Coverage is a measurement indicating what functionalities of the DUV have been tested during a simulation run. By specifying quantitative values for the desired coverage the verification engineer can determine when verification is finished. It uses the bve_cover class (BVE = Brazil-IP Verification Extensions), which is part of the methodology proposed in this paper, see In Figure5, for an example. That class is a In-house creation to be used in our methodology. The Verification Engineering must specify in this class, what functional characteristics must be verified, i.e. what cover criterions must be reached by the verification. In example from Figure 5 we show a cover criterion that only permit to stop the simulation after the sum reach 10 times the number 1, 10 times the number 0 and 10 times the number 2.

The bve_cover class contains a optional progress bars which allow the verification engineer to monitor verification progress during the simulation run.

```
output_checker_cv.begin();
   BVE_COVER_COND(output_checker_cv, 1, 10);
   BVE_COVER_COND(output_checker_cv, 0, 10);
   BVE_COVER_COND(output_checker_cv, 2, 10);
output_checker_cv.end();
```

Figure 5: Functional Cover

The checker is also reponsible for self-checking capability by comparing the results coming from reference model and monitor. This comparison is done at the transaction_level,

The Reference Model receives data from the Source through FIFO(s) and sends data to the Checker through FIFO(s). All data in the Reference Model are transaction_level. The tool generates the required FIFOs and connects them. The functionality from Reference Model must be implemented by the verification engineering. Any compiled object code that can be linked into C++ can be used as reference model. Input transaction data is used as arguments to subroutine or method calls and the output transactions receive their data from the results. Depending on the operating systems used to run the simulator, IPC (inter process call) or RPC (remote procedure call) can also be used to run the reference model.

3. IMPLEMENTATION DETAILS

We have implemented our tool using the SystemC library and SCV. SystemC is based on the C++ programming language and thus it considerably simplifies the creation of a high level environment. On the top of C++ SystemC adds such important concepts as concurrence, events and hardware data types to enable efficient designs. The SCV library improves SystemC capability by providing APIs for transaction based verification, constrained and weighted randomization, exception handling and other verification features.

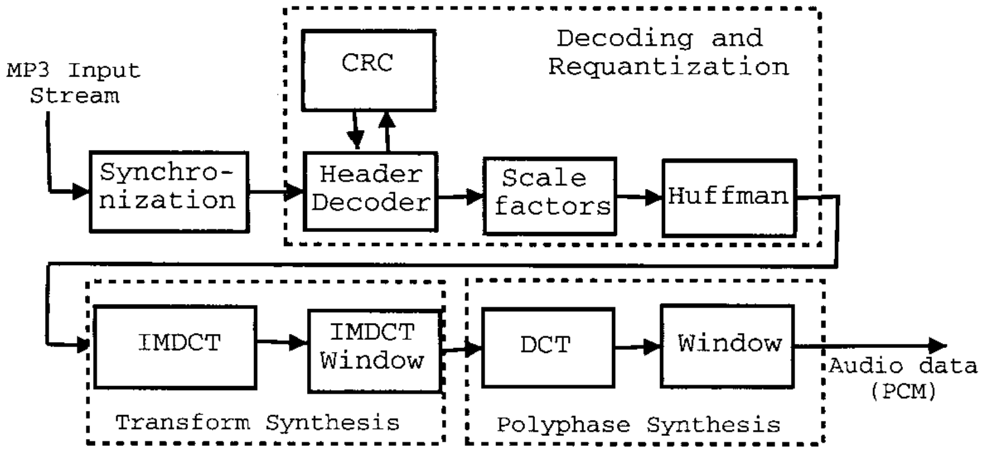


Figure 6: MP3 blocks schema

Furthermore, SCV permits transaction level programming, a methodology that enables a high-level abstraction, reutilization and simulation speedup.

It is well known that providing the design with randomized input data is a very good technique to test the functionality of the design [1]. This comes from the fact that even the rare input cases can be simulated with constrained randomization, leading to a verification coverage that is difficult to obtain by using directly specified inputs. Moreover, in order to verify all states of the design it is required that every important functionality has been tested during simulation. To do this, a functional coverage mechanism is provided that monitors the progress of the verification process. With functional coverage monitoring one knows, at any moment, which percentage of a specified full coverage has already been achieved.

4. APPLICATION EXAMPLE

Our test case DUV is an MP3 decoder implementation, which is part of the BrazilIP project[9]. The entire MP3 decoder project was verified using VeriSC, see Figure 6. We have choosen the window function to show the testbench produced by VeriSC.

The window function is responsible for generating audio data (PCM) from subband samples. The reference model used is from the Libmad library that follows the ISO standard and is open source code with GPL licence.

The next subsection describes the MP3 functional verification steps and points out the most relevant design errors found by the testbench.

4.1 Verification of Window module

The Window function's environment has only two interfaces, an input and an output interface. The source uses floating point to provide random input data to the module. Input data was generated as sets of data pairs with a precision of up to the 9th decimal digit. Random input was created by using the SCV library's class SCV_CONSTRAINT, with SCV_BAG. Handshake to do the interfaces driver-DUV and DUV-monitor was also implemented.

To make the comparison between the Reference Model results and DUV results at the Checker Module, the Root Mean Square method (RMS), was used:

$$\sqrt{\frac{1}{n}\sum_{k=0}^{n-1}(x_k - y_k)}$$

where n is the samples number, x is the reference samples and y is the resulting samples of our module decodification. A function in the Checker was created that permits the comparison between the module outputs. According ISO standards, the RMS must not be higher than $\frac{2^{-15}}{\sqrt{12}}$, furthermore $|x_k - y_k|$ it must not be higher than 2^{-14}. Through this function we could verify if the window outputs were within specifications.

4.2 Results

The verification found three major design errors that were not found during the preliminary simulation which did not use the proposed methodology. The list of relevant mistakes is shown below:

1 In the Finite State Machine (FSM), the reset state was not reinitializing the nt[2][512] vector to zero.

2 The first 15 output blocks caused errors when the reset signal is raised;

3 The module decoded correctly only stereophonic data, but was not capable to decode monophonic data correctly.

Notice that subtle design errors, like error 3 above, could hardly be captured if only simulation or standard verification procedures were used.

By using the VeriSC tool the three designers of the MP3 could considerably speedup the verification of the MP3 design, cutting in half the design time. All errors have been corrected. Verification was repeated and it could not find any more mistakes.

5. CONCLUSION

Verification engineers must employ tools that allow them to do easier, quicker and more reliable functional verification.

In this paper, we propose a new methodology that allows transaction-level, coverage-driven, self-checking and random-constraint functional verification. We have shown a tool, based on SystemC and SCV, that implements this methodology. Furthermore we presented the verification of an MP3 module and showed that, by using the VeriSC tool, the designers could capture very hard design errors, considerably reducing the design time.

6. ADDITIONAL AUTHORS

Additional authors:Valdiney Alves Pimenta(Universidade Estadual de Campinas. Albert Einstein Avenue, 125. Campinas - SP, Brasil. Email: ra005055@ic.unicamp.br).

7. REFERENCES

[1] BERGERON, J., *Functional Verification of HDL models*, Kluwer Academic Publishers, Second Edition, 2002.

[2] Rashinkar, P., Paterson, P., Singh, L., *System-on-a-chip Verification: Methodology & Techniques*, Kluwer Academic Publishers, February, 2001.

[3] Bhasker, J., *A SystemC Primer*, Star Galaxy Publishing, 2002.

[4] FERRANDI, F., RENDINI, M., SCIUTO, D., *Functional Verification for SystemC Descriptions using Constraint solving*, Automation and Test in Europe Conference and Exhibition (DATE'02), p.0704, Paris, March,2002.

[5] REGIMBAL, S., LEMIRE, J.-F., SAVARIA, Y., BOIS, G., ABOULHAMID, M., BARON, A., *Automating Functional Coverage Analysis Based On An Executable Specification*, Proc. of the International Workshop on System-on-Chip for Real-Time Applications, Calgary, June,2003.

[6] DRUCKER,L., *SystemC Verification Library speeds transaction-based verification*, D&R Industry Articles,EEdesign, EEtimes,February, 2003.

[7] FOURNIER, L., ARBETMAN, y., LEVINGER, M., *Functional Verification Methodology for Microprocessors Using the Genesys Test-Program Generator*,Design, Automation and Test in Europe (DATE '99), p.434, Munich, March 09, 1999.

[8] MONACO, J., HOLLOWAY, D., RAINA, R., *Functional Verification Methodology for PowerPC 604 Microprocessor*,33rd Design Automation Conference, DAC 96, Las Vegas.

[9] http://www.brazilip.org