Measuring the Quality of a SystemC Testbench by using Code Coverage Techniques

Daniel Große[1] Hernan Peraza[1] Wolfgang Klingauf[2] Rolf Drechsler[1]

[1] *Institute of Computer Science, University of Bremen, 28359 Bremen, Germany*
[2] *Dept. E.I.S, Technical University of Braunschweig, 38106 Braunschweig, Germany*
[1]{grosse, drechsle}@informatik.uni-bremen.de [2]klingauf@eis.cs.tu-bs.de

Abstract

The system description language SystemC enables to quickly create executable specifications at adequate levels of abstraction for both hardware/software integration and fast design space exploration. Besides the modeling of a system, verification has become a dominant factor in circuit and system design. Since SystemC is a versatile language based on C++, testbenches at different abstraction levels can easily be built. But the fault coverage of a manually developed testbench is hard to quantify. In this paper, an approach for measuring the quality of SystemC testbenches is presented. The approach is based on dedicated code coverage techniques and identifies all the parts of a SystemC model that have not been tested. Experimental results show the applicability of our methodology.

1. Introduction

To cope with the design complexity of hardware/software systems that consist of up to one billion transistors, raising the level of abstraction in modeling has been exercised during the past years in the computer aided design community. In this context, C/C++-based languages have found entrance into industry. Here, the system description language SystemC is the de facto standard and was standardized by the IEEE [13]. Additionally to the inherent SystemC feature of specifying hardware and software in one language the concept of *Transaction Level Modeling* (TLM) [2] is supported by SystemC. TLM allows to describe the communication in a system in terms of abstract operations (transactions).

Besides the modeling aspect the verification – i.e. ensuring the correct functional behavior – is the most challenging problem. Since complete formal verification methods are only applicable to medium sized designs, simulation-based techniques are used most frequently [6, 17]. Here system level languages like SystemC already offer some features for verification and are therefore superior to traditional *Hardware Description Languages* (HDLs). E.g., in SystemC the testbench can easily be integrated as part of the model and all features of C++ can be used for the generation of tests. Also the result analyzer that is typically build to check the response of the *Device Under Verification* (DUV) is a SystemC module. As an add-on for SystemC the *SystemC Verification* (SCV) library has been introduced [15]. Besides advanced verification features like data introspection and transaction recording the SCV library enables constraint-based randomization.

However, all these verification features do not include a measure how thorough the design was executed during the simulation. As the size of the testbench grows the designer needs a reliable feedback about its quality.

In this paper, an approach for SystemC to measure the quality of the testbench is presented. Our analysis is based on dedicated code coverage techniques that we have developed for SystemC models. By exploiting automated code instrumentation based on a SystemC parser, for each test run a coverage report is generated that presents the user all statements in the model that have not been executed during simulation. The report is based on the analysis of the exercised control flow statements. It includes exact source code references to unexecuted code blocks in combination with SystemC specific information like process context and hierarchy information.

The rest of this paper is structured as follows. Related work is described in the next section. In Section 3 we present our approach. We start with the overall flow and continue with a detailed description of the three phases of our approach. Along the way we provide an example to show the effects of each phase. Case studies for two SystemC designs are presented in Section 4. The first design is a RISC CPU and the second design is a TLM-based video processor. Finally, in Section 5 the paper is summarized.

2. Related Work

In software testing code coverage techniques have been used to measure the fraction of code that has been exercised by a test case [1]. From this domain coverage methods have been derived and extended for HDLs. For Verilog or VHDL several approaches and tools exists (for an overview see e.g. [16]). However, to the best of our knowledge no code coverage method to measure the quality of a SystemC testbench has been proposed. Note, that approaches based on standard C++ coverage tools (like e.g. the GNU COVerage tool gcov [7]) have several drawbacks. On the one hand the SystemC kernel is also included in the coverage analysis. On the other hand SystemC specific data like e.g. context information or hierarchy information is only implicitly available and has to be extracted manually.

In the following we present an approach to overcome such limits.

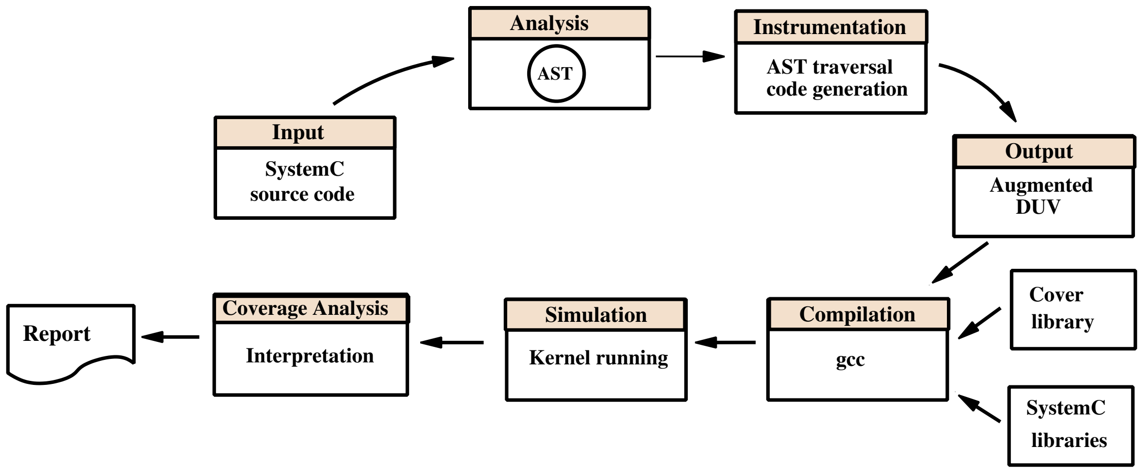

Figure 1. Overall flow of our approach

3. Measuring the Quality of a SystemC Testbench

In this section the code coverage-based approach for measuring the quality of the testbench is introduced. Our approach consists of three phases: SystemC analysis, code instrumentation and coverage analysis. Before the details on the three phases are given the overall flow is presented. Throughout the description of the phases a simple example is used to demonstrate the effects of each phase.

3.1. Overall Flow

The overall flow of our approach is depicted in Figure 1. In the analysis phase the SystemC code of the DUV is parsed, analyzed and transformed into an *abstract syntax tree* (AST) representation. This AST is traversed in the consecutive code instrumentation phase. During the traversal the original SystemC DUV is augmented with SystemC specific code that enables the collection of coverage information during simulation. Then, the rewritten SystemC DUV, the coverage library of our approach and the SystemC libraries are compiled into an executable. By running this executable simulation is performed and the data structures available through our coverage library are filled. Finally, in the coverage analysis phase the collected data is interpreted and the coverage report is generated. By the report the verification engineer is informed which statements have not been executed due to the tests defined in the testbench. This information is presented with exact source code references to unexecuted blocks in the original SystemC DUV including hierarchy. Furthermore the frequency of the execution of statement blocks can be given for further analysis.

In the following we describe the three phases in more detail.

3.2. SystemC Analysis

For the transformation of the SystemC DUV into an AST the front-end from [5, 8] is used that is part of the design environment SyCE [3]. The parser of the front-end was build with PCCTS (Purdue Compiler Construction Tool Set) [14]. PCCTS enables the description of the SystemC syntax in form of a grammar, provides facilities

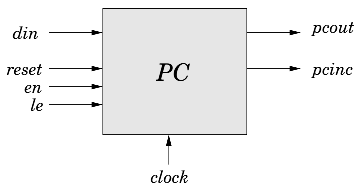

Figure 2. Program counter.

for AST construction and finally generates a parser. Note that the front-end has an exact source code reference including character positions of each token. Therefore, a special C++ preprocessor has been implemented to allow for identification of the SystemC macros before they are expanded. The correct source code information annotated to each node in the AST is very important for our approach. Without this information only a non-reliable feedback for the verification engineer would be possible. In the following the analysis phase is demonstrated by an example.

Example 1 *Since we use a program counter of a RISC CPU also as example for the other phases we give some details on this module. Figure 2 shows the program counter (PC) with all its inputs and outputs. In order to address the 2048 entries of the program memory, the PC has an 11 bit register which holds the address of the current instruction. Output* pcout *holds this address.* pcinc *outputs the address increased by one. An address can be loaded into the PC via the input* din*, if the input* le *(load enable) is set to 1. Using the* reset *signal, the PC can be set to 0. On every positive edge of the* clock *signal the current address is increased if the input* en *(enable) is set to 1. In Figure 3 the method that computes the* `next_state` *of the PC is shown. This method is sensitive to the positive clock.* `pc` *is the internal register of the PC module. Figure 4 depicts a sample of the AST of this method, which has been generated by our tool. Please note that for each AST node only a fragment of the available information is shown. The second number in each line corresponds to the line number of the parsed element.*

As can be seen, the structure of the SystemC program is reflected and this representation is well suited for code instrumentation.

```
 9 void prog_count::next_state(){
10  if (reset.read()){
11   pc = 0; //reset to adress 0
12  }else{
13    if(en.read()){
14      if(le.read()){
15        pc = din; //load address
16      }else{
17        // increase counter
18        pc = pc.read() + 1;
19      }
20    }else{
21      pc = pc.read();
22    }
23  }
```

Figure 3. Parts of the original SystemC DUV

```
10 IF
10  LPAREN
10  ID == "reset"
10  DOT
10      ID == "read"
10  LPAREN
10      RPAREN
10  RPAREN
10  LCURLY
11      ASSIGNEQUAL
11         ID == "pc"
11         OCTALINT
11      SEMICOLON
12      RCURLY
12  ELSE
12  LCURLY
13    IF
13     LPAREN
13        ID == "en"
...
```

Figure 4. AST of next_state method

3.3. Code Instrumentation

In the code instrumentation phase the SystemC DUV is augmented with according instructions to allow for coverage analysis. The main steps in this phase are described in the following.

Coverage Library First, the global variable `cov` is defined that holds an instance of our coverage class `COVER`. This class provides data structures like hash tables for coverage statistics as well as wrapper functions to take care of the control flow inside the methods of the DUV. Furthermore, the class has methods to analyze the collected coverage data and to generate the report for the user.

AST Traversal and Code Instrumentation While traversing the AST, first the member functions that belong to a SystemC module are identified. Then, in each function the conditions of the control flow statements are substituted with wrapper functions. The idea is to perform a call-back during the simulation and thereby notifying the coverage class which control branch has been taken. The following control statements are distinguished: IF, IF/ELSE, SWITCH-CASE, FOR loop, WHILE loop. Next, the wrapper functions are explained.

Wrapper Functions For the IF, IF-ELSE, FOR loop and WHILE loop the condition of the control statement is replaced by a wrapper function call. The arguments of the wrapper functions are:

1. the condition of the control statement (as Boolean and string),
2. the type of control statement,
3. start position and end position of the block(s) that are executed if the control condition evaluates to true/false,
4. file name of the current method,
5. class name if available,
6. current method name,
7. `this` pointer, in case of a member function. The this pointer is used to distinguish between several instances of the same module.

The following example demonstrates the application of a wrapper function for an IF-ELSE control statement.

Example 2 *Consider again the program counter in Figure 3 and focus on the if statement in line 10 and the corresponding else-branch starting in line 12. The condition of the if statement is the expression* `reset.read()`*. This expression is replaced by the wrapper function* `wrapperStatement(...)`*. The instrumented code is depicted in Figure 5. The first and second argument of this function hold the condition as a Boolean and as a string, respectively. The third argument reflects the type of the condition statement – here* `tIFELSE`*. Then, the next four numbers mark the if-block, i.e. the if-block starts in line* `10` *at the absolute character position* `125` *and ends in line* `12` *at character position* `203`*. The next two numbers give the same information for the else-block, but only the end position of the else-block is used; the else-block ends in line* `12` *at character position* `419`*. Then, the file name where the method is implemented (*`prog_count.cc`*), the class name (*`prog_count`*), the method name (*`next_state`*) and the* `this` *pointer are given.*

In a SWITCH-CASE statement at the beginning of each `case` block we instrument a wrapper function that has as additional argument the value of the current case. Note that the approach is able to handle also nested variants of all types of control statements. In the next section the coverage analysis phase is explained.

3.4. Coverage Analysis

After the compilation of the instrumented SystemC code the coverage analysis is executed during simulation. Based on the instrumented wrapper functions the instance of the cover class collects all the coverage data. The main data structures in the cover class are based on *Standard Template Library* (STL) maps. As unique keys the arguments of the wrapper functions are transformed into a string representation. To each coverage point we associate two counters to track the frequency of the evaluation of the

```
#include "label.h"
extern COVER *cov;

#include"prog_count.h"
...
void prog_count::next_state(){
if(cov->WrapperStatement(reset.read(),"
     reset.read()", tIFELSE
     ,10,125,12,203,22,419,"prog_count.cc
     ", "prog_count", "next_state",this))
     {
  pc = 0;
}else{
...
```

Figure 5. Instrumented code of the next_state method

```
<< COVERAGE REPORT >>

IF-ELSE Statement: *IF-BLOCK NOT EXECUTED*
  File name: prog_count.cc
  Class: prog_count
  Instance: pc
  Func. Member: next_state
  Condition: le.read()
  IF start: line 14 pos 246
  IF end:   line 16 pos 322
  count total: 87
  count TRUE: 0 count FALSE: 87
```

Figure 6. Coverage report for program counter.

corresponding condition to true or false. For case statements obviously only one counter is needed. Finally, in the coverage report that is started by a call from `sc_main` after the end of the simulation, the coverage data is analyzed. For IF, IF/ELSE a warning is generated if the condition was always true/false and thus a block was never executed. In case of FOR loops or WHILE loops we inform the user if the condition was false all the time and therefore the loop body was skipped. For SWITCH-CASE statements each case is identified that was never activated. In total this allows to argue about the quality of the tests defined by the testbench. If blocks have been identified that have been never executed these blocks are dead code or the testbench has to be improved.

In the following example the results of the coverage analysis are shown for the program counter.

Example 3 *A testbench has been written for the program counter shown in Figure 3. The testbench includes three tests. We applied our approach for this example. The automatically generated coverage report is shown in Figure 6. As can be seen the scenario to load a value into the program counter by setting load enable to 1 was not executed. We added another test for this behavior and thereby closed this gap.*

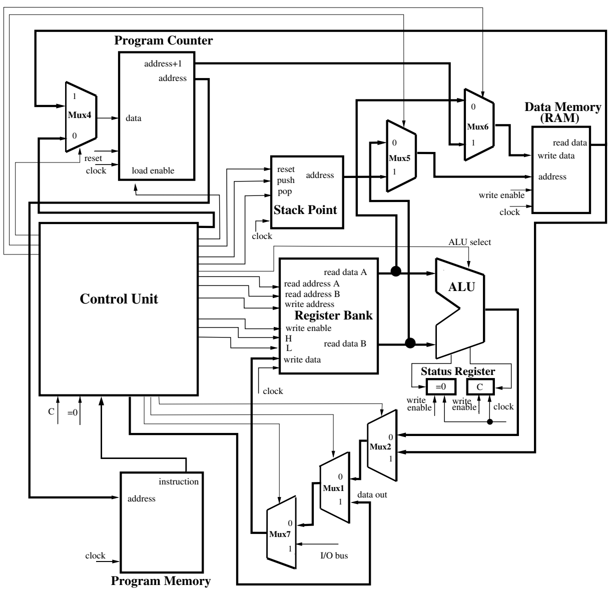

Figure 7. RISC CPU including data and program memory

4. Case Studies

In this section we apply the approach to two examples. The first example is a hardware oriented model, a RISC CPU is considered. The second example is a system for color region recognition in video data.

4.1. Hardware Model: RISC CPU

Before we apply our method to the RISC CPU the basic data of the CPU is briefly reviewed (see [9] for more details).

Specification

In Figure 7 the components of the RISC CPU are shown. The CPU has been designed as a Harvard architecture. The data width of the program memory and the data memory is 16 bit. The size of the program memory is 4 KByte and the size of the data memory is 128 KByte. The length of an instruction is 16 bit. We briefly describe the five different classes of instructions in the following: 6 load/store instructions, 8 arithmetic instructions, 8 logic instructions, 5 jump instructions and 5 other instructions. For the RISC CPU a compiler has been implemented which generates object code from an assembler program. This object code runs on the SystemC model, i.e. the model of the CPU executes an assembler program.

Testbench Quality

Based on successful simulation of each component the designer starts with the simulation at the system level. For this purpose usually a high-level testbench is created that enables a black-box test of the design. For the CPU such a testbench corresponds to the execution of a set of assembler programs including the analysis of the simulation results. In the following we describe how the high-level testbench was created and how this process was improved by our approach. The SystemC model of the RISC CPU

```
        LDL R[6], 0
        LDH R[6], 0
        LDL R[2], 0
        LDH R[2], 0
        LDD R[3],R[2]
loop1:
        ADD R[2],R[2],R[1]
        LDD R[4],R[2]
        ADD R[5],R[4],R[0]
        SHR R[5],R[5]
        XOR R[6],R[4],R[5]
        STO R[2], R[6]
        SUB R[3],R[3],R[1]
        JNZ loop1
        HLT
```

Figure 8. Assembler program for gray code

was automatically instrumented with code to analyze coverage. The following non-trivial assembler program was formulated to test the CPU.

Example 4 *The assembler program shown in Figure 8 converts a set of numbers into gray-code. The gray code encodes numbers such that in the binary encoding adjacent numbers have a hamming distance of 1. The number n of elements to be converted is given in the data memory at address 0. After clearing the register* R[6] *and* R[2], *n is loaded into register* R[3]. *Then, in the loop each single number is converted. The idea is to invert each bit if the next higher bit of the input value (read from the data memory into register* R[4]*) is set to one. Therefore the input is shifted by one and a bitwise XOR operation is performed. The result* R[6] *of the conversion is stored in the data memory to the same position as the input.*

After simulation of the gray code program on the CPU our approach reported unexecuted code fragments in the following modules: `stack_point`, `mux4`, `mux5`, `mux6`, `mux7` and `alu`. The handling for the cases of push and pop operations in the `stack_point` module was not tested, since the inputs from the control unit to this module have been zero during the complete simulation. To test this behavior another program that uses push and pop instructions has to be added.

For the multiplexor modules we found that in the method `do_select` which describes the functionality of a multiplexor only the ELSE-block for the select condition was simulated. For the CPU this observation corresponds to the fact that the select inputs of the multiplexors have been zero all the time and thus only one data input was routed to the multiplexor output. As can be seen in Figure 7 all multiplexors belong to the data path of the CPU. To also test the effects on the CPU in case of data coming through the other input, a different data path has to be activated. The multiplexor `mux5` is part of the stack pointer data path and thus was tested by using stack pointer operations (see above). For `mux4` and `mux6` the alternative data path is activated by adding a program that uses subroutine calls. For `mux7` we set the select input to one by an additional program that uses I/O instructions.

In case of the ALU several CASE statements of the main SWITCH statement have not been executed since not all operations of the ALU are activated by the considered assembler programs. Therefore we created another program to check to remaining arithmetic operations.

In total by adding additional assembler programs to the testbench the quality of the testbench was improved. Here our approach supported the verification engineer by directly pointing to untested functionality of the RISC CPU.

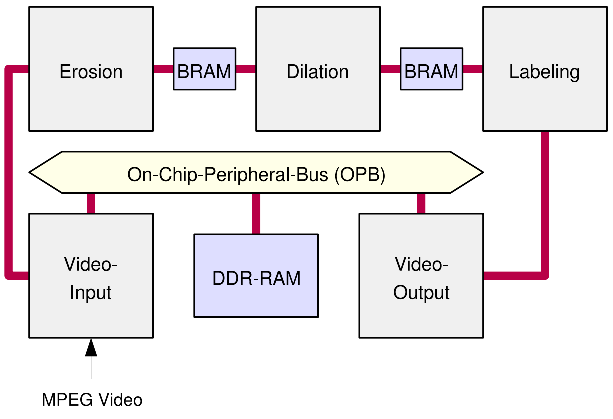

Figure 9. Color region recognition schematic

4.2. High-Level Model: Color Region Recognition

In the second example we applied our approach to a high-level SystemC model of a video processor System-on-Chip. In contrast to the RISC CPU (which has been implemented as an RTL design), this model resides at the transaction-level of abstraction.

Specification

The configurable model *EmVid* consists of a set of SystemC cores that can be integrated to build a video processor. For video input and output, abstract TLM channels are used. The video processing IP cores use the *SystemC High-level Interface Protocol* (SHIP) [10] for data exchange over these channels. Communication with the main memory (DDR RAM) is established by ST‘s TAC protocol [12]. In the following, we consider a System-on-Chip for color region recognition that is based on EmViD cores. The system processes video frames in real-time and draws rectangles around detected regions. A high-level schematic of the system is shown in Figure 9. The system has been configured as a pipelined architecture and for the connection of the DDR RAM an IBM CoreConnect *On-Chip Peripheral Bus* (OPB) is used. The complete transaction-level interconnect (including an OPB simulation model) is set up using the GreenBus TLM fabric [11]. EmViD can be found on [4].

The video processing starts by reading in an MPEG video as video input. Then, dilation and erosion is performed. In the labeling stage the regions are recognized and the rectangles are added. Afterwards the core outputs the image to a display.

Testbench Quality

As a concrete application we decided to detect skins in the video data. We set the color range for the recognition accordingly. The system segmentates the processed video data in the labeling phase. Therefore adjacent pixels are analyzed and the image is partitioned into a set of regions using the defined color information.

Table 1. Video processor execution traces

Config	# ex. video	# ex. detect.	FPS video	FPS detect.	Comment
Bus only model 1	500	500	24.98	24.98	ascending priority
Bus only model 2	451	872	22.55	43.60	higher detection priority
Mixed bus/pipeline model 1	500	500	24.98	24.98	lower pipeline priority
Mixed bus/pipeline model 2	500	999	24.98	49.90	higher pipeline priority

In the overall video processor system the high-level testbench consists of the video data (coming from video files or a camera). We applied our approach to the system. We simulated the system with different video files and observed that depending on the video data different parts of the system have not been executed. For example, in the `morph_segm` module (labeling phase) the segmentation algorithm checks the minimum region size with an IF-condition. For video data that contains no skins or very small areas no regions are detected. Here, our approach presents directly the SystemC file with the exact source code position of the never executed block(s). Note that this improves the debugging during the development of such high-level models significant. Moreover, analyzing the results of nested control structures – which are used in the segmentation algorithm – our approach helps the verification engineer to test the design thoroughly.

Further Design Analysis

During the analysis of the video processor model, we also experimented with different communication architecture configurations for the design. As one might expect, some architectures are better suited than others to meet efficiency requirements such as a given frame rate. In particular, when connecting all components to a shared bus with fixed-priority scheduling (here, the OPB), the overall video processing performance highly depends on the priority allocation.

We utilized the ability of our coverage analysis to count the number of executions for the various processes in the model in order to identify the location of communication bottlenecks in design configurations with poor frame rates. Table 1 presents some results of the experiments. Lines 1 and 2 show the frame rates we got with a bus-only model. While in line 1, the bus access priorities were assigned in ascending order according to the sequence of video processing stages in the model, in line 2 we assigned a higher priority to the region detection components than to the video display datapath. As expected, the frames per second processed for region detection goes up, but as an unintentional side effect due to higher bus workload, the number of video frames displayed per second drops down.

Lines 3 and 4 show the results we achieved with a mixed bus/pipeline model as depicted in Figure 9. Here, we could considerably increase the video display frame rate by just swapping the bus access priorities of two components.

5. Conclusions

In this paper, we have presented an approach to measure the quality of a testbench for a SystemC design. The approach is based on dedicated code coverage techniques using a SystemC front-end. Thus, a reliable feedback for untested parts of the design are presented to the user. This data includes exact source code information in combination with SystemC specific information, like process context and module hierarchy. In summary, our approach helps to create a high quality testbench. The experiments showed that our approach is suitable for both RTL and TLM designs. Moreover, the TLM example revealed that our analysis methodology also can support design space exploration.

References

[1] B. Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., 1990.

[2] L. Cai and D. Gajski. Transaction level modeling: an overview. In *CODES+ISSS '03*, pages 19–24, 2003.

[3] R. Drechsler, G. Fey, C. Genz, and D. Große. SyCE: An integrated environment for system design in SystemC. In *IEEE International Workshop on Rapid System Prototyping*, pages 258–260, 2005.

[4] *EmViD: Embedded Video Detection.* http://www.greensocs.com/GreenBench/EmViD.

[5] G. Fey, D. Große, T. Cassens, C. Genz, T. Warode, and R. Drechsler. ParSyC: An Efficient SystemC Parser. In *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, pages 148–154, 2004.

[6] R. S. French, M. S. Lam, J. R. Levitt, and K. Olukotun. A general method for compiling event-driven simulations. In *Design Automation Conference*, pages 151–156, 1995.

[7] http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[8] C. Genz and R. Drechsler. System exploration of SystemC designs. In *IEEE Annual Symposium on VLSI*, pages 335–340, 2006.

[9] D. Große, U. Kühne, and R. Drechsler. Hw/sw co-verification of embedded systems using bounded model checking. In *Great Lakes Symp. VLSI*, pages 43–48, 2006.

[10] W. Klingauf. Systematic transaction level modeling of embedded systems with SystemC. In *Design, Automation and Test in Europe*, pages 566–567, 2005.

[11] W. Klingauf, R. Günzel, O. Bringmann, P. Parfuntseu, and M. Burton. Greenbus: a generic interconnect fabric for transaction level modelling. In *Design Automation Conf.*, pages 905–910, 2006.

[12] S. Microelectronics. TAC: Transaction Accurate Communication. *http://www.greensocs.com/TACPackage*, 2005.

[13] Open SystemC Initiative, http://www.systemc.org. *SystemC 2.1 Language Reference Manual*, 2005.

[14] T. Parr. *Language Translation using PCCTS and C++: A Reference Guide*. Automata Publishing Co., 1997.

[15] SystemC Verification Working Group, http://www.systemc.org. *SystemC Verification Standard Specification Version 1.0e*.

[16] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design and Test of Computers*, 18(4):36–45, 2001.

[17] J. Yuan, C. Pixley, and A. Aziz. *Constraint-based Verification*. Springer, 2006.