Improvements for Constraint Solving in the SystemC Verification Library

Daniel Große
Institute of Computer Science
University of Bremen
28359 Bremen, Germany
grosse@informatik.uni-bremen.de

Rüdiger Ebendt
German Aerospace Center
Institute of Transport Research
12489 Berlin, Germany
ruediger.ebendt@dlr.de

Rolf Drechsler
Institute of Computer Science
University of Bremen
28359 Bremen, Germany
drechsle@informatik.uni-bremen.de

ABSTRACT

For verification of complex system-on-chip designs often constraint-based randomization is used. This allows to simulate scenarios that may be difficult to generate manually. For the system description language SystemC the SystemC Verification (SCV) Library has been introduced. Besides advanced verification features like data introspection and transaction recording the SCV library enables constraint-based randomization for SystemC models. However, the SCV library has two disadvantages that restrict their practical use: There is no support of bit operators in SCV constraints and the SCV constraint solver cannot guarantee a uniform distribution of the constraint solutions. In this paper we provide a detailed analysis of these problems and present solutions that have been integrated in the library.

Categories and Subject Descriptors: J.6 [Computer-aided Engineering]: Computer-aided design (CAD)

General Terms: Verification

Keywords: SystemC, Constraint-based Randomization, SystemC Verification Library

1. INTRODUCTION

Today circuit and system design is a very challenging task. Due to the increasing usage of circuit and systems in all kinds of devices – ranging from cell phones to safety critical systems – functional verification has become an important issue. Since complete formal verification methods are only applicable to medium sized designs, simulation-based techniques are used most frequently [3, 13].

In *directed simulation* explicitly specified stimulus pattern are applied over a number of clock cycles to the design in order to stimulate a certain functionality and the response is compared with the expected result. However, directed simulation only checks single scenarios. Since these scenarios have to be generated manually, this is a very time consuming and expensive task.

In order to overcome this limitation *random pattern simulation* is used to generate random stimulus patterns for the design. E.g. random address and data is computed to verify communication over a bus.

GLSVLSI'07, March 11–13, 2007, Stresa-Lago Maggiore, Italy.

To reduce the amount of time for the specification of simulation scenarios *constraint-based random stimulus generation* has been introduced (see e.g. [13]). For the simulation only those stimulus pattern are generated that satisfy certain given constraints. By this, the random stimulus generation process is controlled. The resulting stimuli allow to test scenarios that may be difficult to generate manually. This helps to uncover bugs that may otherwise remain undetected especially because corner cases are not excluded.

In the context of system description languages, like SystemC, the *SystemC Verification* (SCV) library [12] has been introduced for constraint-based random stimulus generation and was used successfully in industrial verification projects (see e.g. [9, 4]). The SCV library allows for system level verification using constrained randomization. However, the SCV library has two major disadvantages that restrict their practical use. On the one hand in the constraints no bit operators are supported. On the other hand the constraint solver does not fulfill the important requirement that the constraint solutions must be uniformly distributed.

In this paper we analyze these two problems and describe improvements that overcome these limitations. The paper is structured as follows. Section 2 starts with an introduction of SystemC, the SCV library and BDDs, the basic data structure that is used in the SCV to represent constraints. Then, in Section 3 the improvements for the SCV library are presented. First, new bit operators for SCV constraints are introduced. In the second part we explain how a uniform distribution across all constraint solutions can be guaranteed. Finally, the paper is summarized in Section in 4.

2. PRELIMINARIES

In this section, first SystemC and the SCV library are described. Next the basic underlying data structure that is used by the SCV library to represent constraints is briefly reviewed.

2.1 SystemC

As a C++ class library SystemC [11] enables modeling of systems at different levels of abstraction starting at the functional level and ending at a cycle-accurate model. The well-known concept of hierarchical descriptions of systems is transferred to SystemC by describing a module as a C++ class. Furthermore, fast simulation is possible at an early stage of the design process and hardware/software co-design can be carried out in the same environment. For details on SystemC we refer to [5, 7].

2.2 SystemC Verification Library

The first version of the SCV library was introduced in December 2002 as an open source class library [12, 8, 6]. It

layers on top of the system description language SystemC and adds tightly integrated verification capabilities to SystemC. In the following the main features of the SCV library are summarized:

- Data introspection for SystemC and C++ data types
  - Manipulation of arbitrary data types
  - Manipulation of data objects without compile time information
- Transaction API
  - Transaction monitoring and recording
  - Basis for debugging, visualization and coverage
- Constraint-based stimulus generation for SystemC and C++
  - High quality pseudo random generator
  - Integrated constraint solver based on BDDs

Since this paper presents improvements for constraint solving using the SCV library in the following we briefly review the basic underlying data structure used by the constraint solver.

2.3 Binary Decision Diagrams

As is well-known a Boolean function $f : \mathbf{B}^n \rightarrow \mathbf{B}$ can be represented by a *Binary Decision Diagram* (BDD) which is a directed acyclic graph where a Shannon decomposition

$$f = \overline{x}_i f_{x_i=0} + x_i f_{x_i=1} \quad (1 \leq i \leq n)$$

is carried out in each node.

A BDD is called *ordered* if each variable is encountered at most once on each path from the root to a terminal node and if the variables are encountered in the same order on all such paths. A BDD is called *reduced* if it does not contain isomorphic subgraphs nor does it have redundant nodes. Reduced and ordered BDDs are a canonical representation since for each Boolean function the BDD is uniquely specified [2]. In the following, we refer to reduced and ordered BDDs for brevity as BDDs.

EXAMPLE 1. *In Figure 3 a BDD for the function* $f = \overline{x}_1 x_2 + x_1 x_2 x_3$ *is shown. The BDD is shown using complement edges [1]. These allow to represent both a function and its complement by the same node, modifying the edge pointing to the node instead. Therefore the BDD only contains the 1-terminal.*

2.4 Constraints in the SCV Library

In the SCV library constraints are declared as classes. This allows an object-oriented way to manage constraints using hierarchy and inheritance. In detail a constraint is derived from the `scv_constraint_base` class. The data to be randomized is specified as `scv_smart_ptr` variables.

EXAMPLE 2. *An example of an SCV constraint is shown in Figure 1. The name of the constraint is* my_constraint. *Here, the two unsigned integer variables a and b are randomized. The conditions on the variables a and b are defined by expressions in the* SCV_CONSTRAINT() *macro.*

Internally a constraint in the SCV library is represented by the corresponding characteristic function, i.e. the function is true for all solutions of the constraint. This characteristic function of a constraint is represented as a BDD. As BDD package CUDD [10] is used in the SCV library.

```
1  struct my_constraint : public
       scv_constraint_base {
2    scv_smart_ptr<sc_uint<32> > a,b;
3
4    SCV_CONSTRAINT_CTOR(my_constraint) {
5      SCV_CONSTRAINT( a() > 100 );
6      SCV_CONSTRAINT( b() == 0 );
7    }
8  };
```

Figure 1: Example constraint

```
1  struct constraint : public
       scv_constraint_base {
2    scv_smart_ptr<sc_uint<32> > a,b;
3
4    SCV_CONSTRAINT_CTOR(constraint) {
5     SCV_CONSTRAINT(a().range(1,3) == 5);
6     SCV_CONSTRAINT(b()[10] == 1);
7    }
8  };
```

Figure 2: Example constraint with bit operators

3. IMPROVEMENTS FOR CONSTRAINT SOLVING

After a detailed analysis of the SCV constraint solver two disadvantages were found that restrict the practical use. In the following two sections we describe the problems and the solutions.

3.1 Bit Operators

Constraint expressions over variables to be randomized can only use the following operators:

- Arithmetic operators: `+, -, *`
- Relational operators: `==, !=, >, >=, <, <=`
- Logical operators: `!, &&, ||`

As can be seen there is no support for bit operators in the SCV constraint solver. However, bit operators are very important for the verification engineer during the specification of constraints. Bit operators allow for simpler and more compact formulations of complex constraints. In detail the following bit operators have been implemented:

1. Bitwise and: `a() & b()`
2. Bitwise or: `a() | b()`
3. Bitwise not: `~a()`
4. Bit-select: `a()[i]` for constant i
5. Slice-select: `a().range(x,y)` for constant x and y

EXAMPLE 3. *In Figure 2 an example constraint that uses bit operators is shown. Note that these kinds of constraints can otherwise not be written in such a simple and compact way.*

For the implementation first in the class `scv_expression` the according operators were overloaded and new member functions were added. The class `scv_expression` is used for the internal representation of the constraint expressions in form of an expression tree. In such a tree leaf nodes are variables or constants and non-terminal nodes are marked with operators.

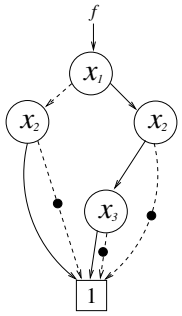

Figure 3: BDD for $f = \overline{x}_1 x_2 + x_1 x_2 x_3$

The class `_scv_expr` is used to store the BDD representation of an `scv_expression`. For the construction of the BDD in this class each bit operator has to be mapped to the according BDD synthesis operations. For example in case of a "bitwise and" the resulting bit vector is computed by the BDD-AND operation for each bit of the two input vectors. Of course there are several special cases like different length of vectors, different data types etc. that have to be taken into account.

3.2 Uniform Distribution

The uniform distribution of the solutions of constraints is a very important aspect for the quality of a constraint solver. However, we observed that the solutions are not always uniformly distributed. This problem occurs in scenarios of high practical relevance. If variables are fixed to a certain value for constraint solving, i.e. these variables are disabled for randomization then the solutions computed by the constraint solver are not uniformly distributed across the set of all possible solutions. Later we will illustrate this phenomenon by a simple example. Before describing our remedy we explain the constraint solving process of the SCV library in more detail.

The constraint solver works on individual bits when solving constraints. As explained in Section 2 for a constraint, a BDD representation is computed. The constraint solver generates a solution of a constraint by using the BDD that represents the constraint. For this purpose the algorithm starts at the root node and traverses the BDD down to the 1-terminal. A path starting from the root and ending at the 1-terminal determines the values of the variables along the path. These values correspond to a solution of the constraints since the BDD is the characteristic function of the constraint. One could assume that choosing the 0- or 1-assignment for a Boolean variable with a probability of 50% guarantees a uniform distribution. However, the following observation shows that this is not true. As explained above all constraint solutions are paths to the 1-terminal starting from the root node. But during the BDD traversal some sub-BDDs can have more paths to the 1-terminal than other sub-BDDs. Thus, if a sub-BDD with fewer paths is selected this leads to an overweighting of the fewer represented solutions. This is illustrated by the following example.

EXAMPLE 4. *In Figure 3 the BDD for the function $f = \overline{x}_1 x_2 + x_1 x_2 x_3$ is shown, where dashed lines are used for the 0-assignment and a dot on an edge represents a complement edge (i.e. the function below is inverted). If during the BDD traversal of the function the 1-edge of the root node is choosen there is exactly one path to the 1-terminal. If instead the 0-edge is chosen, the reached sub-BDD has* two *paths to the 1-terminal: In the non-reduced BDD there is a node marked with x_3 that is reached by assigning $x_1 = 0$ and $x_2 = 1$; the 1-edge of this node as well as the uncomplemented 0-edge point to the 1-terminal.*

In total, the probabilities following the intuitive traversal algorithm are shown in Table 1.

Table 1: Probabilities for solutions

x_1	x_2	x_3	probability
0	1	0	25%
0	1	1	25%
1	1	1	50%

This example demonstrates that the probability for choosing the 1-edge of the root node should be corrected to 33% instead of 50%. By this a uniform distribution across all solutions is achieved.

In the SCV constraint solver a special weighting algorithm is implemented to guarantee the uniform distribution of all solutions. In a pre-processing step the BDD of all initial constraints is traversed and the correct probabilities are computed for each node. The basic idea of the recursive weighting algorithm is to compute weights of the else- and then-child of a node while taking into account whether nodes have been removed due to BDD reduction rules. Based on the weights a probability is assigned to each BDD node. Then, for the generation of values – one constraint solution is picked uniformly distributed across all solutions of the constraint – the computed probabilities of the pre-processing step are used during the BDD traversal.

The SCV constraint solver calls the weighting algorithm only once at the beginning for the initial BDD that represents the constraints. Thus, the SCV constraint solver is not able to handle simplifications like e.g. fixing variables to a certain value. In this case the BDD that represents the constraints is modified due to the simplification but the probabilities are not updated. This causes a non-uniform distribution of the constraint solution. We provide an example for this observation. For simplicity we only use one single constraint in the following example. Note that the presented technique is not restricted to this case. An arbitrary number of constraints as well as derived constraints including the hard/soft constraint mechanism of the SCV library are fully supported. After the example we give some technical details for solving the problem.

EXAMPLE 5. *Consider the constraint in Figure 4. This constraint specifies that $a + b = c$ and c is fixed to* 99 *(see lines 7 and 8). The distribution shown in Figure 5 was the result from running this constraint 100,000 times in the SCV constraint solver. As can be seen there is a strong bias of the solutions in the middle part.*

One has to overcome several difficulties while correcting this behavior. This is due to the design of the SCV library which divides the functionality of handling the BDD-based representation of constraints roughly into two parts, i.e. one global constraint manager object and the respective constraint objects (one for each constraint specified). On the one hand, it may seem natural to direct all BDD-related tasks via one dedicated constraint manager object (which is encapsulated in a class called `_scv_constraint_manager` and creates a CUDD manager object at start). On the other hand, a closer inspection unveils serious flaws in the centralized design:

The SCV constraint manager maintains the number of BDD variables necessary for representing the least recently used constraint object. However, this number is reseted (i.e., "forgotten") as soon as a new constraint object is cre-

```
1  struct triangle_c : public
       scv_constraint_base {
2    scv_smart_ptr<sc_uint<7> > a,b;
3    scv_smart_ptr<sc_uint<8> > c;
4
5    SCV_CONSTRAINT_CTOR(triangle_c) {
6      SCV_CONSTRAINT( a() + b() == c() );
7      c->disable_randomization();
8      *c = 99;
9    }
10 };
```

Figure 4: Triangle constraint

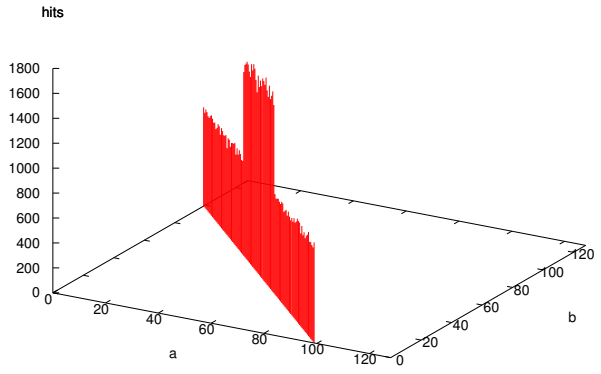

Figure 5: Distribution for $a + b = 99$ with original SCV

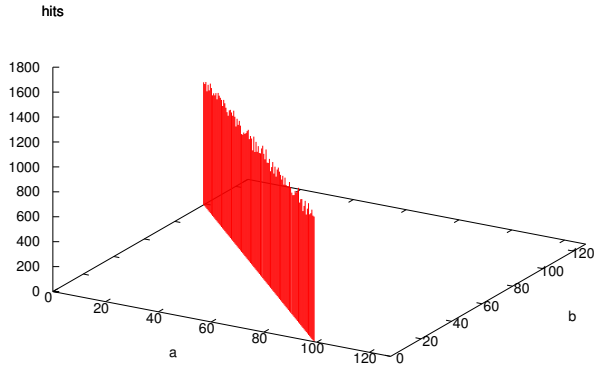

Figure 6: Distribution for $a + b = 99$ with improved SCV

ated. Hence if a previous constraint is simplified, this number is not available anymore. This is a problem since the weighting algorithm crucially depends on this number. There are similar problems with the information contained in two hash tables stored at the constraint manager object: the tables `nodeHashP` and `nodeWeightHash` hold the probability information for all nodes of the BDD representing a constraint and the according weighting information, respectively. At the time of simplification of a previous constraint, the data stored in the table needs to be cleared which is not done by the SCV system.

We did a complete redesign of the constraint management classes that also solved these problems. Furthermore we put more intelligence into the constraint objects. E.g. now every constraint object is capable of giving back a pointer to its BDD representation via a method `getBddNodeP`. We have preserved backwards compatibility and full functionality/structure of the SCV interfaces (e.g., the possibility of overloading `C++` virtual member functions like `scv_constraint_base::next` which triggers the next random assignment of the constraint variables). In order to achieve this, it was necessary to give the simplified BDD to the constraint object in several methods of the class `_scv_constraint_manager`, e.g. `assignRandomValue`, as well as in several internal utility routines called from other code within the SystemC verification standard, e.g. in `_scv_set_value`.

The result of our redesign now is a tight integration of the weighting algorithm and the BDD synthesis operations. After structural modifications in the interfaces and correct initialization of internal data structures now the weighting algorithm is called after a simplification. Thus, the weights and probabilities are recomputed and a uniform distribution is achieved. In Figure 6 the result for the constraint from Example 5 is shown again for 100,000 times calling the SCV constraint solver. As can be seen a uniform distribution among all solution was established.

4. CONCLUSIONS

Constraint-based randomization using the SCV library is crucial for complex verification tasks in todays system-on-chip designs. In this paper, two major problems of the SCV library have been tackled by a careful redesign and several extensions of the SCV core. As a result, all issues that previously existed due to non-uniform distribution of generated stimuli have been resolved. Moreover, as a benefit for the verification engineer during constraint creation, the constraint specification language has been extended in expressiveness. Both means a significant improvement in practicality of the SystemC Verification Library.

5. ACKNOWLEDGMENTS

This research work was supported in part by the German Federal Ministry of Education and Research (BMBF) in the project URANOS under the contract number 01M3075.

6. REFERENCES

[1] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *Design Automation Conf.*, pages 40–45, 1990.

[2] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[3] 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.

[4] D. Große, R. Siegmund, and R. Drechsler. Processor verification. In P. Ienne and R. Leupers, editors, *Customizable Embedded Processors*, pages 281–302. Elsevier, 2006.

[5] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[6] C. N. Ip and S. Swan. A tutorial introduction on the new SystemC verification standard. http://www.systemc.org. White paper, 2003.

[7] W. Müller, W. Rosenstiel, and J. Ruf, editors. *SystemC Methodologies and Applications*. Kluwer Academic Publishers, 2003.

[8] J. Rose and S. Swan. SCV randomization version 1.0. 2003.

[9] R. Siegmund, U. Hensel, A. Herrholz, and I. Volt. A functional coverage prototype for SystemC-based verification of chipset designs. In *9th European SystemC User Group Meeting at Design, Automation and Test in Europe*, 2004.

[10] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.3.0*. University of Colorado at Boulder, 1998.

[11] Synopsys Inc., CoWare Inc., and Frontier Design Inc., http://www.systemc.org. *Functional Specification for SystemC 2.0*.

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

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