Reachability Analysis for Formal Verification of SystemC*

Rolf Drechsler

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

Daniel Große

Institute of Computer Science
Albert-Ludwigs-University
79110 Freiburg im Breisgau, Germany
grosse@informatik.uni-freiburg.de

Abstract

With ever increasing design sizes, verification becomes the bottleneck in modern design flows. Up to 80% of the overall costs are due to the verification task. Formal methods have been proposed to overcome the limitations of simulation approaches. But these techniques have mainly been applied to lower levels of abstraction. With more and more design complexity the need for hardware description languages with a high level of abstraction becomes obvious.

We present a formal verification approach for circuits described in SystemC, an extension of C that allows the modeling of hardware. An algorithm for reachability analysis is proposed and a case study of a scalable bus arbiter cell is given.

1. Introduction

Nowadays complex circuits can only be described on a higher level of abstraction. *Hardware Description Languages* (HDLs), like VHDL and Verilog, are used to describe hardware on the *Register-Transfer Level* (RTL) or on even higher levels based on C/C++-like languages [9]. One very promising candidate for these descriptions is SystemC [11], since it combines the hardware aspects with the ability of fast simulation. Several successful implementations based on SystemC have recently been reported.

But as in other HDLs, verification is also a major issue. Even though simulation can often be carried out faster than for a corresponding design on the gate level, pure simulation is not sufficient to guarantee the correct circuit behavior (see e.g. [1]). So far, several verification approaches for SystemC have been reported, but all of them are based on simulation [10, 6] and do not consider the paradigms of formal techniques, i.e. to prove the correctness of a circuit behavior.

As has been observed by many authors [5, 3, 7, 4, 13, 8], formal verification is very closely related to reachability analysis of *Finite State Machines* (FSMs), if the underlying circuit is modeled appropriately. This directly results from the fact that for each state in the circuit it has to be checked whether it is reachable from the initial (or reset) state to argue about its correct or erroneous behavior.

In this paper we present a reachability analysis algorithm for hardware systems described in SystemC. It is based on symbolic computations using *Binary Decision Diagrams* (BDDs). A case study of a scalable bus arbiter shows that the algorithm works very well also for complex designs with a high sequential depth.

The paper is structured as follows: In Section 2 BDDs are briefly described and the basics on reachability analysis are introduced. We describe how circuits are modeled in SystemC. The formal verification approach is shown in Section 3 and the experimental results are given in Section 4. Finally, the results are summarized.

2. Preliminaries

2.1. 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.

For functions represented by reduced and ordered BDDs efficient manipulations are possible [2]. In the following, only reduced and ordered BDDs are considered and for brevity these graphs are called BDDs.

2.2. Reachability Analysis of Sequential Circuits

Definition 1 *Let $w = \{w_1, \ldots, w_m\}$ denote a set of input variables, $x = \{x_1, \ldots, x_n\}$ a set of present state variables, and $y = \{y_1, \ldots, y_n\}$ a set of next state variables.*

*This work was supported in part by DFG grant DR 287/8-1.

ReachabilityAnalysis$(I(x), \tilde{T}_\delta(x,y))$ {
  $Reached(x) = I(x)$
  do {
    $ReachedBefore(x) = Reached(x)$
    $Im(y) = \exists x\ (\tilde{T}_\delta(x,y) \cdot Reached(x))$
    $Im(x) = Im(y)|_{y \leftarrow x}$
    $Reached(x) = Reached(x) \cup Im(x)$
  } while $(Reached(x) \neq ReachedBefore(x))$
}

Figure 1. Sketch of reachability analysis

Further let $\delta : \mathbf{B}^{m+n} \rightarrow \mathbf{B}^n$ *be the next state transition function of a sequential circuit.*

1. *The* transition relation $T_\delta : \mathbf{B}^{m+2\cdot n} \rightarrow \mathbf{B}$ *is defined by*

$$T_\delta(w,x,y) = 1 \Leftrightarrow \delta(w,x) = y,$$

   i.e. $T_\delta(w,x,y) = 1$ *iff state* y *can be reached in exactly one transition from state* x *when input* w *is applied.*

2. *The* image *of a set* $S \subseteq \mathbf{B}^n$ *according to* T_δ *is given by* $Image(S)(y) = \exists w, x\ (T_\delta(w,x,y) \cdot S(x))$.

Now the set of all reachable states can be computed as a fixpoint iteration of image computation. It starts with the set of all initial states $I(x)$ and the transition relation $\tilde{T}_\delta(x,y) = \exists w\ T_\delta(w,x,y)$ because the existential quantification of the input variables w has only to be done once. The process stops as soon as no new states can be found. In Figure 1 a sketch of the method is shown.

2.3. Modeling Circuits in SystemC

In this section we briefly describe the main features of SystemC for modeling a circuit [10]:

1. Modules: Modules are the basic container objects which can include ports, data and function members, and other modules. Thus, a hierarchical design description becomes possible.

2. Processes: Processes are used to describe the functionality. They are declared as special functions of modules and can be reactive to any input signal or to a clock signal.

3. Ports: Through ports a module can send or receive data. SystemC supports single-directional and bi-directional ports. Ports determine the direction of data from one module to another.

4. Signals: Signals represent physical wires and interconnect modules. Signals carry data without information about directions.

5. Clocks: Clocks are special signals and the timekeepers during simulation.

```
SC_MODULE(AndGate) {
 sc_in<bool> in1;
 sc_in<bool> in2;
 sc_out<bool> out;
 void entry();

 SC_CTOR(AndGate) {
  SC_METHOD(entry);
  sensitive << in1 << in2;
 }
 void end_of_elaboration() {
  symb->reportAND(name(),out,in1,in2);
 }
};

void AndGate::entry() {
  out.write(in1.read() && in2.read());
}
```

Figure 2. AND gate

Because SystemC is an extension of C++, all C++ concepts can be used to describe the behavior of a circuit. Various levels of abstraction are possible in SystemC.

At RTL we define as basic gates AND, OR, NOT and flipflop, which enables a description of any sequential circuit. As an example the AND gate is shown in Figure 2. The important command for our approach can be seen in the method `end_of_elaboration()`. The relevance is discussed in Section 3.1.

3. Reachability Analysis in SystemC

To be able to compute the set of reachable states as introduced we have to construct BDDs for the outputs and the transition functions of the flipflops. Therefore we first need a complete description of the circuit and second a topological order of the underlying netlist.

3.1. Netlist and Topological Order

We start with an outline on how the description of the circuit is transformed to a model that can be used for reachability analysis.

After instantiating and properly connecting all modules in `sc_main()`, primary inputs of the circuit have to be registered via `registerInputs()` to the class `Symb`. In this class all data structures and methods for reachability analysis are collected. An important task to be solved was the link between the defined SystemC basic gates and the internal representation of this gates in class `Symb`. Intuitively, one might try to use the constructor `SC_CTOR` of a module for this purpose, because only the type of gate and its interconnection to other gates is needed. The problem here was that at that time the constructor of a SystemC module is called not every structure of SystemC is totally initialized, e.g. the interface of a port is

```
int sc_main(int argc, char* argv[]) {
   // make clocks, signals,
   // connect modules, etc.
   ...
   registerInputs();
   sc_initialize();
   symb->startSymb();
   ...
}
```

Figure 3. Routine `sc_main()`

not bound. So it is impossible to get the pointer of the bool value in '`sc_in<bool> in1`' of the AND gate as shown in Figure 2. This pointer is needed to determine the input gate. A solution to this problem is the already shown method `end_of_elaboration()`, which is a virtual method called for all modules, channels, and ports after elaboration, i.e. just before simulation starts. By default this method is empty, but can be redefined to perform static checking that cannot be executed during elaboration. By calling `sc_initialize()` or `sc_start()` ¿from `sc_main()` the SystemC scheduler is initialized which causes a preparation of all modules and different thread/process types. Thus, all gates are then known to the class `Symb`, because they are reported after the elaboration due to method `end_of_elaboration()` of each basic gate, where the corresponding report method, e.g. `reportAND(name(),out,in1,in2)` (see Figure 2), is called. As a result each gate of the SystemC definition becomes an element of an internal hashtable of `Symb`. As unique id's the pointers of gates to their inputs and output data are used. In Figure 3 the important commands to be used in `sc_main()` are shown again. By calling `symb->startSymb()` the control is passed to the class `Symb`: First for each gate in the hashtable the input gates are searched and assigned to it. In the next step a topological order of the netlist is computed. For this purpose a simple algorithm propagates a marking from the primary inputs to the outputs.

3.2. Set of Reachable States

Based on the topological order the BDDs for the circuit outputs and the transition functions of the flipflops are built. Hence for each flipflop a state variable and for each input a variable has been created, i.e. a new BDD node. Now computing the set of reachable states becomes simple. After constructing the transition relation the fixpoint iteration of image computation is done.

The relevant functions of class `Symb` are shown as pseudo code in Figure 4. In our implementation we used the STL library [12] for data structures, like hashtables, vectors, etc.

```
class Symb {
...

public:
  // called in end_of_elaboration() of
  // corresponding SystemC-gate
  void reportAND(name,out,in1,in2);
  void reportOR(name,out,in1,in2);
  void reportNOT(name,out,in);
  void reportFlipFlop(name,out,in);

  // reachability analysis
  void startSymb() {
    for (each gate) {
      inputs = findInputGates(gate);
      interConnect(gate,inputs);
    }
    computeTopologicalOrder();
    computeTransitionRelation();
    Reached = ResetStates;
    do {
      ReachedBefore = Reached;
      Reached = Reached
                  or Image(Reached);
    } while (Reached != ReachedBefore);
  }
};
```

Figure 4. Class `Symb`

4. Case Study

In this section experimental results are given. The algorithm has been implemented in C++. All runtimes are given in CPU seconds on an AMD Athlon 800MHz with 512 MByte of main memory. As a benchmark for our experiments we considered a scalable bus arbiter. This circuit is often used for experiments in formal verification (see e.g. [8, 10]). The n-cell arbiter circuit is defined in SystemC based on the introduced basic gates. In the upper part of Figure 5 a single arbiter cell is shown, whereas the composition to an n-cell arbiter is given in the lower part.

The results are given in Table 1. The number of arbiter cells is shown in column *Cells*, and column *States* gives the number of reachable states computed by our algorithm. The third column reports the size of the transition relation $T_\delta(w, x, y)$ in number of BDD nodes. In the last two columns the runtime in CPU seconds and the memory needed are given, respectively. As can be seen, with increasing number of arbiter cells the memory and runtime needed grows exponentially. Nevertheless, for up to 11 cells the states can be computed exactly allowing to formally argue about the reachability and thus the complete verification of the underlying FSM.

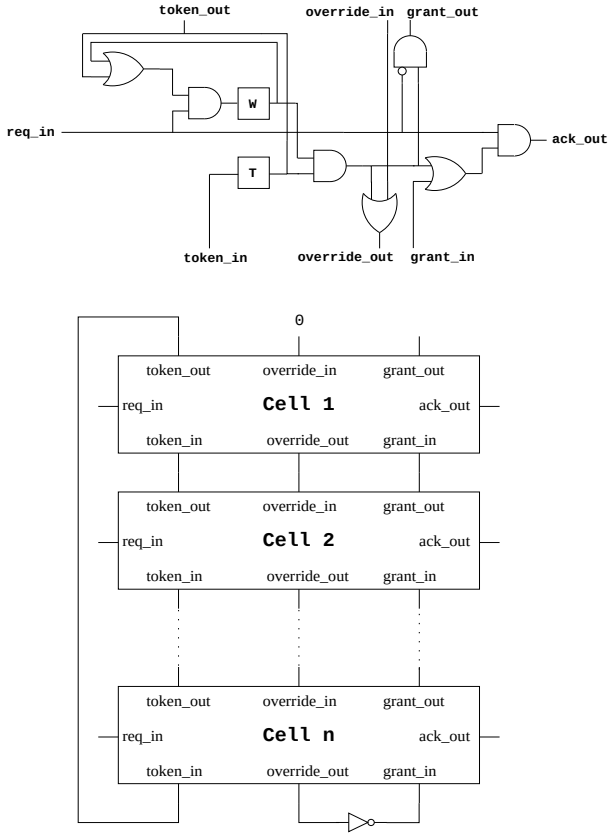

Figure 5. The arbiter circuit

Table 1. Computation of reachable states

Cells	States	Size of $T_\delta(w,x,y)$	time	MByte
2	8	57	0.01	4.1
3	24	245	0.01	4.1
4	64	1,005	0.01	4.2
5	160	4,061	0.02	4.4
6	384	16,317	0.11	5.1
7	896	65,405	0.78	8.1
8	2048	261,885	3.95	15.2
9	4608	1,048,061	29.87	33.1
10	10240	4,193,277	169.36	132.1
11	22528	16,775,165	1332.18	460.2

5. Conclusions

We presented an algorithm to compute the set of reachable states for circuits given as a SystemC description. The algorithm is based on symbolic computations using BDDs. A case study of a scalable bus arbiter has shown that the technique is applicable to larger designs and allows the formal verification of circuits specified in SystemC.

It is focus of current work to include this technique in a complete verification flow including property checking. In contrast to the approach in [10] this will not be based on simulation, but on formal proof techniques that allow to guarantee the correct functional behavior. In addition, further case studies are needed to get a better understanding of the applicability of the approach.

References

[1] B. Bentley. Validating the Intel Pentium 4 microprocessor. In *Design Automation Conf.*, pages 244–248, 2001.

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

[3] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite–state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, 1986.

[4] O. Coudert and J.C. Madre. A unified framework for the formal verification of sequential circuits. In *Int'l Conf. on CAD*, pages 126–129, 1990.

[5] D. Dempster and M. Stuart. *Verification Methodology Manual - Techniques for Verifying HDL Designs*. Teamwork International, 2001.

[6] F. Ferrandi, M. Rendine, and D. Scuito. Functional verification for SystemC descriptions using constraint solving. In *Design, Automation and Test in Europe*, pages 744–751, 2002.

[7] G. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publisher, 1996.

[8] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.

[9] R. Gupta (moderator). IEEE design and test roundtable on C++-based design. *IEEE Design & Test of Comp.*, pages 115–123, 2001. May-June.

[10] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on multi-valued ar-automata. In *Design, Automation and Test in Europe*, pages 742–748, 2001.

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

[12] Silicon Graphics Computer Systems. Standard template library programmer's guide. *http://www.sgi.com/tech/stl*, 1999.

[13] The VIS Group. VIS: A system for verification and synthesis. In *Computer Aided Verification*, volume 1102 of *LNCS*, pages 428–432. Springer Verlag, 1996.