FORMAL VERIFICATION OF LTL FORMULAS FOR SYSTEMC DESIGNS

Daniel Große *Rolf Drechsler*

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

ABSTRACT

To handle today's complexity, modern circuits and systems have to be specified at a high level of abstraction. Recently, SystemC has been proposed as a language that allows a fast simulation on a high level of abstraction and an efficient realization on RTL. To guarantee the correct behavior of a design, a concise verification methodology has to be developed.

We present the first formal verification approach for SystemC that allows to prove the correctness of properties specified in linear temporal logic (LTL). In contrast to simulation-based techniques, completeness can be ensured. Our proof engine is based on symbolic manipulation, and a case study of a scalable bus arbiter shows the efficiency of the approach.

1. INTRODUCTION

While classical approaches to circuit design make use of hardware description languages (HDLs), like VHDL or Verilog, there is a strong interest in C-like description languages [5]. These languages allow for fast simulation in an early stage of the design process. Furthermore, hardware/software co-design can be performed in the same system environment. One of the most popular languages of this type is SystemC [7].

To cope with today's complexity, a concise verification methodology has to be developed, since more than 70% of the overall design costs are due to verification. Formal verification has shown to be a very promising approach [4], and in equivalence checking these tools have become the state-of-the-art. First approaches to checking the behavior of a circuit based on (bounded) model checking [3, 8] have been reported. But so far, all verification approaches for SystemC are based on simulation only (see e.g. [6, 2]).

In this paper we present a new approach to reason about the behavior of SystemC designs based on formally verifying properties specified in linear temporal logic (LTL). The approach considers synchronous sequential circuits modeled in SystemC at gate level. First, the output functions and transition functions of the underlying finite state machine (FSM) are computed. Then a symbolic reachability analysis of the FSM is carried out. Finally, proving of an LTL formula is translated to a satisfiability problem using the transition and output functions and the set of reachable states. Unbounded LTL formulas can be proved, since the complete set of reachable states is known. A case study of a scalable bus arbiter shows the efficiency of the approach.

2. PRELIMINARIES

2.1. Modeling Circuits in SystemC

SystemC is a C++ class library which allows to describe both software and hardware. Here we focus on a description of hardware based on logic gates and for this define basic gates, like AND, OR, NOT, and flipflop. By this any sequential circuit can be modeled. As an example the AND gate is shown in Figure 1. It can be seen that there is clear distinction between the interface (first three lines), the functionality (doAnd), and the sensitivity list. There is also room for user defined methods like `end_of_elaboration()`, which is explained below.

2.2. Reachability Analysis for a SystemC Circuit

For performing the fix-point iteration of image computation to obtain the set of all reachable states, the output and transition functions of the underlying circuit are needed. With the virtual method `end_of_elaboration()`, which can be seen as an extension of the basic gate (see e.g. Figure 1), we are able to identify a gate and its interconnection to other gates in a SystemC design. As a result of such a method call, done by the SystemC simulation kernel just before simulation starts, every instance of a gate is reported to a data structure in the class `Symb`. This is realized by `reportAND` in the `end_of_elaboration()` method. By starting the simulation the control is passed to `Symb` where the BDDs for the output and the transition functions are built, the transition relation is constructed and finally the fix-point iteration is carried out. For details of reachability analysis in SystemC see [1].

3. PROPERTY CHECKING

This section describes how the proof engine for LTL formulas described for a SystemC circuit works. For syntax and semantics of LTL we refer to [6]. The main constructs are

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

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

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

Figure 1: AND gate

given in Figure 2, i.e. beside the atomic propositions, several operators exist to describe events in the next step (X), in the future (F), or properties that have to hold generally (G). Some examples of formulas are described in detail in the case study below. In contrast to standard LTL we also support time bounds for the temporal operators, i.e. $F_{[m,n]}$ and $G_{[m,n]}$. By this, we can also restrict the argumentation to a limited number of time frames, since this often simplifies the proof process (see [8]).

In the following we describe the overall flow of our verification approach. First, assume that an LTL formula f contains only bounded operators, i.e. the upper bound of the operators F or G are finite. Then f is checked as follows (see Figure 3):

1. While parsing f the examination window $[t_{min}, t_{max}]$ of f is determined and a list $list_{BDD-Op}$ is created, which contains the corresponding BDD operations to be performed later in step 3.
2. The circuit is unrolled t_{max} times.
3. Based on $list_{BDD-Op}$ and the unrolled circuit the Boolean function f^* is constructed. Checking the satisfiability of f^* proves or disproves f.

In step 2 the unrolling of a circuit makes it possible to observe signal values of the circuit at different points in time. Unrolling means identifying the current state variables with the previous next state variables of the circuit. The unrolling process is illustrated in Figure 4, where I_j is the set of primary inputs, O_j the set of outputs, and S_t the set of states each at time j. All functions are represented by BDDs.

The construction of f^* in step 3 can be achieved by carrying out the BDD operations of $list_{BDD-Op}$ where atoms of f are replaced by the corresponding BDD nodes computed in step 2. Now the true support of f^* can only contain variables of $I_t, I_{t+1}, \ldots, I_{t_{max}}, S_t$.

```
// atomic proposition:
LTL& prop(const sc_signal<bool>&);

// neXt-operator
LTL& X(ulong m, LTL& f);
LTL& X(LTL& f);

// Generally-operator
LTL& G(ulong m, ulong n, LTL& f);
LTL& G(ulong n, LTL& f);
LTL& G(LTL& f);

// Future-operator
LTL& F(ulong m, ulong n, LTL& f);
LTL& F(ulong n, LTL& f);
LTL& F(LTL& f);

// logical operators
// conjunction:
LTL& operator&(LTL&, LTL&);
// disjunction:
LTL& operator|(LTL&, LTL&);
// implication:
LTL& operator>(LTL&, LTL&);
// negation:
LTL& operator~(LTL&);
```

Figure 2: Representation of *LTL*-formulas

```
class Symb {
...

public:
  // call in sc_main()
  void startSymb() {
    ...
    // reachable states have been computed
    // δ, λ and Reached are available
    // property checking:
    t_min = t_max = 0;
    clearList(list_BDD-Op);
    parseLTL(f,t_min,t_max,list_BDD-Op);
    // list_BDD-Op and the interval [t_min,t_max]
    // have been computed
    unrollCircuit(δ,λ,t_max);
    prove(f);
  }
};
```

Figure 3: Pseudo-code of the proof engine in class `Symb`

In our model we assume that each circuit has an initial state s_0. Therefore, a bounded LTL formula f is considered to be arguing starting from this initial state. Now the LTL formula f holds, iff

$$(\exists I_t, I_{t+1}, \ldots, I_{t_{max}} \neg f^*) \wedge s_0 \equiv 0.$$

f^* is negated because we are interested in assignments of f^*

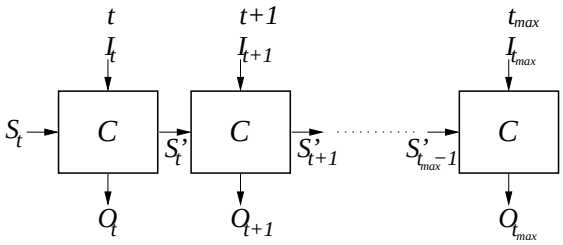


Figure 4: Unrolling of a sequential circuit

under which f^* does *not* hold. If the left side of the equation is zero (that is the empty state set) the formula holds. Otherwise the initial state is a counterexample for f.

Now consider an LTL formula of the form Gf, where f is bounded. Then Gf can be proved like above, but instead of the conjunction with the initial state, the conjunction is built with the set of reachable states, i.e. Gf holds, iff

$$(\exists I_t, I_{t+1}, \ldots, I_{t_{max}} \neg f^*) \wedge Reached \equiv 0.$$

The reachable state set is used, because f has to hold in every reachable state.

Using the equivalence $Ff \equiv \neg G \neg f$ we are also able to show, whether a formula Ff evaluates to true or not. Ff holds, iff

$$(\exists I_t, I_{t+1}, \ldots, I_{t_{max}} f^*) \wedge Reached \not\equiv 0.$$

If this is the case, we have identified at least one reachable state, for which f holds.

4. CASE STUDY: SCALABLE ARBITER

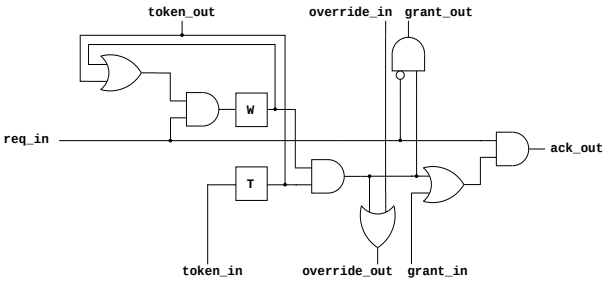
In this section experimental results are given. The algorithm has been implemented in C++. All runtimes are given in CPU seconds on an AMD Athlon 800MHz with 512 MByte of main memory. As a benchmark for our experiments we considered a scalable bus arbiter. This circuit is often used for experiments in formal verification (see e.g. [3, 6]). The n-cell arbiter circuit is defined in SystemC based on 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.

For the n-cell arbiter we consider the following three formulas specified in LTL:

1. *Mutual exclusion*: Two output signals of the arbiter can never become 1 at the same time:

$$G(\bigwedge_{i \neq j}^{n} \neg(ack_i \wedge ack_j))$$

2. *Liveness*: Each request req_i is confirmed by an acknowledge ack_i within $2 \cdot n$ time frames:



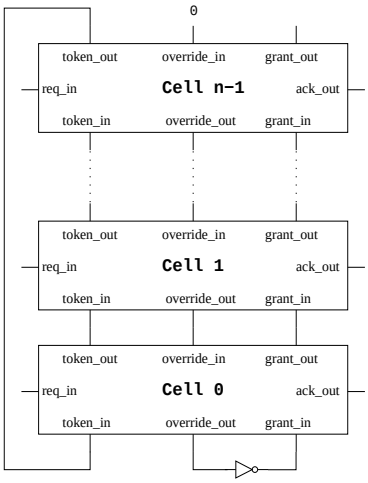


Figure 5: The arbiter circuit

$$G(req_i \rightarrow F_{[0,2n-1]} ack_i)$$ [1]

3. *Conservativeness*: The acknowledge signal ack_i is only set, if there was a request req_i:

$$G(ack_i \rightarrow req_i)$$

The results are shown in Table 1, 2 and 3, respectively. In each table, the first column denotes the number of cells of the arbiter. Then, the number of states is given, followed by information on time in CPU seconds and space in MByte. For measuring the time, we distinguish between the construction process and the proof part.

As can be seen, dependent on the number of cells and the chosen property, runtime and space significantly differ. But for all cases the properties can be proven for up to 9 cells, while mutual exclusion and conservativeness even work for up to 200 cells.

[1] To prove this formula, the environment must guarantee that the request signals are persistent, i.e. in total we get the complete formula $G(G_{[0,2n-1]}(req_i \rightarrow (\neg ack_i \rightarrow X req_i)) \rightarrow (req_i \rightarrow F_{[0,2n-1]} ack_i))$.

Table 1: Results for proving *mutual exclusion*

cells	states	time FSM	time proof	space inc.
2	8	0.01	0.01	<0.01
3	24	0.01	0.01	<0.01
4	64	0.01	0.01	<0.01
5	160	0.04	0.01	<0.01
6	384	0.06	0.01	<0.01
7	896	0.09	0.01	<0.01
8	2048	0.15	0.01	<0.01
9	4608	0.13	0.01	<0.01
10	10240	0.24	0.01	<0.01
11	22528	0.25	0.07	<0.01
12	49152	0.26	0.01	<0.01
20	$2.10 \cdot 10^7$	1.26	0.25	0.05
50	$5.63 \cdot 10^{16}$	38.04	4.85	0.23
100	$1.27 \cdot 10^{32}$	618.49	415.21	0.60
150	$2.14 \cdot 10^{47}$	3424.10	1199.93	10.71
200	$3.21 \cdot 10^{62}$	13751.07	12832.90	5.34

Table 2: Results for proving *liveness*

cells	states	time FSM	time proof	space inc.
2	8	0.01	0.01	0.01
3	24	0.01	0.01	0.08
4	64	0.02	0.61	0.18
5	160	0.06	7.07	1.12
6	384	0.08	98.46	4.78
7	896	0.09	1600.19	33.52
8	2048	0.14	8360.09	92.84
9	4608	0.19	57276.80	354.06

5. CONCLUSIONS

We presented a formal verification tool for proving formulas given in linear temporal logic (LTL) reasoning over circuits specified in SystemC. This is the first verification approach for SystemC that allows to prove the correctness of properties, while all previous approaches are based on simulation. A case study of a scalable arbiter circuit has shown the efficiency of the approach.

It is focus of current work to improve the performance of the proof engine by incorporating alternative solving techniques, like SAT or ATPG.

6. ACKNOWLEDGMENT

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

Table 3: Results for proving *conservativeness*

cells	states	time FSM	time proof	space inc.
2	8	0.01	0.01	<0.01
3	24	0.01	0.01	<0.01
4	64	0.01	0.01	<0.01
5	160	0.05	0.01	<0.01
6	384	0.07	0.01	<0.01
7	896	0.09	0.01	<0.01
8	2048	0.13	0.01	<0.01
9	4608	0.16	0.01	<0.01
10	10240	0.18	0.01	<0.01
11	22528	0.26	0.01	<0.01
12	49152	0.28	0.01	<0.01
20	$2.10 \cdot 10^7$	1.30	0.01	<0.01
50	$5.63 \cdot 10^{16}$	33.11	0.01	<0.01
100	$1.27 \cdot 10^{32}$	587.28	0.01	<0.01
150	$2.14 \cdot 10^{47}$	3492.05	0.01	<0.01
200	$3.21 \cdot 10^{62}$	13089.62	0.01	<0.01

7. REFERENCES

[1] R. Drechsler and D. Große. Reachability analysis for formal verification of SystemC. In *Euromicro Symposium on Digital System Design (DSD'2002)*, pages 337–340, 2002.

[2] 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.

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

[4] R. Drechsler (moderator). IEEE design and test roundtable on formal verification: current use and future perspectives. *IEEE Design & Test of Comp.*, pages 105–113, 2002. Sept-Oct.

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

[6] 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.

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

[8] P.F. Williams, A. Biere, E.M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Computer Aided Verification*, volume 1855 of *LNCS*, pages 124–138. Springer Verlag, 2000.