CheckSyC: An Efficient Property Checker for RTL SystemC Designs

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

***Abstract*— To cope with the increasing complexity of today's circuits and systems new design methodologies are needed. Modeling at higher levels of abstraction and hardware/software integration become more and more important. A language that offers this features is SystemC. But besides efficient modeling, the correct functional behavior has to be ensured.**

In this paper we present the property checker *CheckSyC* for SystemC descriptions on the Resigter Transfer Level (RTL). A SystemC design and a temporal property are converted into a satisfiability (SAT) problem. If the SAT problem is unsatisfiable the property holds. To demonstrate the efficiency of CheckSyC different designs are studied.

I. Introduction

The steadily increasing complexity of modern circuits and systems creates many new challenges for today's design process. While classical circuit design approaches are based on dedicated hardware description languages, like e.g. Verilog or VHDL, modeling on a higher level of abstraction on the one hand and easier integration of software on the other hand becomes more and more important.

A language that addresses these issues is SystemC [1], [2]. As a C++ class library SystemC enables modeling of systems at different levels of abstraction starting at the functional level and ending at a cycle-accurate model. For hardware modeling the SystemC library adds concepts like e.g. timing and concurrency. This allows hardware/software co-design in the same environment. Furthermore fast simulation at an early stage of the design process becomes possible.

In the meantime it has been observed that assuring the correct functional behavior becomes the dominating factor of a successful design. Today in many circuit design projects already up to 80% of the overall design costs are due to verification. As alternatives to classical simulation formal methods have been proposed [3]. The main idea of formal approaches is to prove the functional correctness instead of simulating some vectors. In equivalence checking formal tools are state-of-the-art [4]. In case of SystemC, examples of simulation based techniques are [5], [6]. First formal approaches to check the behavior of a circuit description in SystemC have been reported in [7], [8]. But these approaches only covered a small fraction of the SystemC language and suffered from complexity issues.

In this paper we propose an efficient property checking approach for SystemC designs described on the register-transfer level (RTL). The property checker has been implemented as the tool *CheckSyC*[1]. In contrast to previous approaches (see e.g. [8]), where a gate level based description was required, our approach works on a higher level of abstraction and can also process complex statements. This is due to a new generic front-end [9]. The front-end reads in a SystemC description and builds a representation of the design in form of an Abstract Syntax Tree (AST). The AST is then transformed in consecutive translation steps into a Finite State Machine (FSM) representation. The FSM representation of the SystemC design together with the property to be proven is translated to a Boolean decision problem, which is then solved using a SAT solver. The technique we use is a variant of Bounded Model Checking (BMC) [10]. Experimental results demonstrate that the new approach is very efficient and compared to the BDD-based approach from [8] gives speed-ups of several orders of magnitude.

II. Preliminaries

In the following circuits and systems are modeled in SystemC. Therefore, first, a short overview on SystemC is given. Then the formalism for specification of temporal properties is described.

A. SystemC

The main features of SystemC for modeling a system are based on the following:

- Modules are the basic building blocks for partitioning a design. A module can contain processes, ports, channels and other modules. Thus, a hierarchical design description becomes possible.
- Communication is realized with the concept of interfaces, ports and channels. An interface defines a set of methods to access channels. Through ports a module can send or receive data and access channel interfaces. A channel serves as a container for communication functionality, e.g. to hide communication protocols from modules.
- Processes are used to describe the functionality of the system, and allow expressing concurrency in the system. They are declared as special functions of modules and can be sensitive to events, e.g. an event on an input signal.
- Hardware specific objects are supplied, like e.g. signals, which represent physical wires, clocks, and a set of data-types useful for hardware modeling.

[1] *CheckSyC* is a complete verification environment that beside property checking (PC) also supports equivalence checking and simulation based verification. In the following we restrict ourselves to the description of the PC component.

Besides this, SystemC provides a simulation kernel. The functionality is similar to traditional event-based simulators. Note that a SystemC description can be compiled with a standard C++ compiler to produce an executable specification.

B. Property Language

Describing temporal properties for verification can be done in many different ways, since there exist several languages and temporal logics. In the following we use a notation similar to the property checker from Infineon Technologies AG (see e.g. [11], [12] for more details). A property consists of two parts: a list of assumptions (*assume part*) and a list of commitments (*proof part*). An assumption/commitment has the form

```
    at t+a: expression;
or  during[t+a,t+b]: expression;
or  within[t+a,t+b]: expression;
```

where `t` is a time point, and `a`, `b` $\in \mathbb{N}$ are offsets. If all assumptions hold, all commitments in the proof part have to hold as well. Since `a` and `b` are finite a property argues only over a finite interval which is called *observation window*.

Example 1: The property `test` says that whenever signal x becomes 1, two clock cycles later signal y has to be 2.

```
theorem test is
assume:
  at t: x = 1;
prove:
  at t+2: y = 2;
end theorem;
```

In general a property states that whenever some signals have a given value, some other (or the same) signals assume specified values. Of course it is also possible to describe symbolic relations of signals. Furthermore the property language allows to argue over time intervals, e.g. that a signal has to hold in a specified interval. This is expressed by using the keywords `during` and `within`, whereas `during` states that the expression has to hold all the times in the interval and with `within` the expression has to hold at least once in the specified interval. Also a set of advanced operators and constructs is provided to express complex constraints more easily.

III. Property Checking

In this section we present the property checker CheckSyC. For a SystemC design and a set of properties the proposed approach works as follows (see also Figure 1):

1) The design is transformed into an internal FSM representation.
2) A single property and the FSM representation is translated into a BMC problem.
3) The BMC problem is checked for satisfiability to decide if the property holds or not.

These steps are now discussed in more detail.

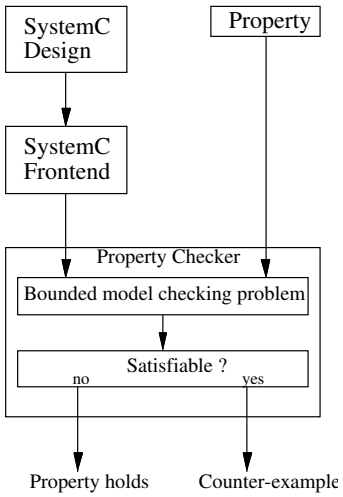

Fig. 1. Property checking work flow

A. SystemC Frontend

The frontend is based on a parser for SystemC [9] which has been developed to be generic in order to allow an application in different areas, like e.g. verification or visualization.

The parser produces an easy-to-process representation of a SystemC design in form of an Abstract Syntax Tree (AST). The tool PCCTS (Purdue Compiler Construction Tool Set) [13] was used to build the parser. PCCTS enables the description of the syntax of SystemC by a grammar, provides facilities for AST construction and finally generates a parser. In consecutive translation steps the AST of a SystemC design is transformed into a FSM representation.

For the transformation of an RTL SystemC description into the corresponding FSM representation SystemC is restricted to a synthesizable subset of possible C++ and SystemC constructs (see e.g. [14]). To prevent difficulties already known from high-level-synthesis C++ features like dynamic memory allocation, pointers, recursions or loops with variable bounds are not allowed. In the same way some SystemC constructs have no direct correspondence on the RTL and are excluded, like e.g. SystemC channels. For channels that obey certain restrictions the FSM transformation can be extended by providing a library of RTL realizations.

Supported are all other constructs that are known from traditional hardware description languages. This comprises different operators for SystemC-datatypes, hierarchical modeling or concurrent processes in a module. Additionally, the `new`-operator is allowed for instantiation of submodules to allow e.g. for a compact description of scalable designs.

B. SAT Formulation of Property Checking

We now describe how a property checking problem formulated on top of the property language introduced in the previous section can be translated into a satisfiability problem. The initial sequential property checking problem is converted into a combinational one by unrolling the design, i.e. by identifying the current state variables with the previous next state variables of the circuit.

A BMC instance of a property p arguing over the finite interval $[t, t+c]$ for a design D is given by:

$$b = \bigwedge_{j=0}^{c-1} T_\delta(\ i(t+j), s(t+j), s(t+j+1)\ )\ \wedge$$
$$\neg\ p(\ i(t), s(t), o(t), \ldots, i(t+c), s(t+c), o(t+c)\ )$$

with

- $i(t) = (i_1^t, \ldots, i_m^t)$ inputs at time point t,
- $s(t) = (s_1^t, \ldots, s_n^t)$ states at time point t,
- $o(t) = \lambda(i(t), s(t))$ outputs at time point t and
- T_δ the transition relation.

The BMC instance b depends only on the states $s(t)$ and the inputs $i(t), \ldots, i(t+c)$. It is unsatisfiable if for all states $s(t)$ and all input sequences $i(t), \ldots, i(t+c)$ the property p over the interval $[t, t+c]$ holds for the design D. If b is satisfiable a counter-example for the property p has been found.

In the rest of this section we explain the realization of the presented techniques for the property checker. CheckSyC takes the FSM representation of the SystemC design and a property as input. Then the property is translated into an expression using only inputs, states and outputs of the SystemC design annotated with time points. The unrolled FSM representation and the property expression are converted into a bit-level representation. Here hashing and merging techniques for minimization are used. The bit-level representation is given to the SAT solver zChaff [15] which has been integrated into CheckSyC. In case of a counter-example a waveform in VCD format is generated to allow for an easy debugging.

IV. Experiments

All experiments have been carried out in the same system environment on a Athlon XP 2800 with 1 GByte of main memory. A run time limit of 1 CPU hour has been set. In the following all given run times do not include the time needed for generation of the FSM representation from a SystemC description, since this has to be done only once for each design. Thus, the resulting FSM representation of a design is stored in a file.

In a first example we studied a FIR-filter of scalable width. Scalable are the number of coefficients and the bit-width of data. A block-level diagram of the FIR-filter is shown in Figure 2. Incoming data is stored in a shift register (`d[0],...,d[n-1]`), coefficients (`c[0],...,c[n-1]`) are stored in a register array. The result is provided at the output `dout`. The SystemC description contains one process to create the shift-register and another process that carries out the calculations. The FIR-filter uses a saturation arithmetic, when the size of the output exceeds a maximum. The coefficients are provided by an array of constants. In Figure 3 the most complex property `calc` for the FIR-filter is shown. With `calc` the correctness of the calculation of a FIR-filter instance with $n = 8$ and an input/output bit-width of 8 is proven. The exact calculation is assigned to the variable `exact`. With the `prev` operator values of variables of previous time points can be accessed. In total it is proven that the data output is set to the maximum or computes the exact value under the assumption of no reset. In Table I the results for three different

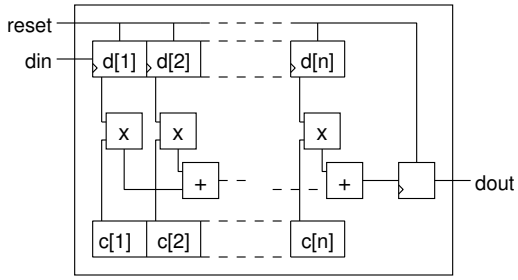

Fig. 2. FIR-filter: Block level diagram

```
theorem calc is
freeze:
  exact = (
   c[0]*prev(din,2) + c[1]*prev(din,3) +
   c[2]*prev(din,4) + c[3]*prev(din,5) +
   c[4]*prev(din,6) + c[5]*prev(din,7) +
   c[6]*prev(din,8) + c[7]*prev(din,9) )@t+9;
assume:
  during[t,t+8]: reset = 0;
prove:
  at t+9: if (exact > MAX) then
            dout = MAX
          else
            dout = exact
          end if;
end theorem;
```

Fig. 3. Property `calc` for FIR-filter

properties of the FIR-filter are given. Besides `calc` a property for the reset and the shifting of data have been formulated. In the first column the name of the property is given. The next two columns provide information about the SAT instance, i.e. the number of clauses and literals, respectively. In the last column the overall CPU time needed is reported. As can be seen the reset and shift property can be proven very fast. Even the complex property `calc` can be proven in less than 500 CPU seconds.

As a second example we studied a scalable hardware realization of the bubble sort algorithm. The SystemC description is shown in Figure 4. This module implements the sort algorithm for eight data words. The bit size of each data word is determined by a `typedef` (line 1). Notice that the approach from [8] did not support constructs, like e.g. typedefs or for-loops. The formulated property `sorted` states that the resulting sequence is ordered correctly, i.e. that the value of an output is greater or equal compared to values at outputs with smaller indices. In Table II the results are given for this property and increasing bit sizes of data words (column *Bit size*). The next two columns again provide information about the SAT instance. In the last column the overall CPU time

TABLE I
Results for FIR-filter

Property	Clauses	Literals	Time(sec)
reset	3382	7600	0.84
shift	3941	8903	0.89
calc	29373	66175	467.19

```
1  typedef sc_uint<4> T;
2  SC_MODULE(bubble) {
3    sc_in<T>  in[8];
4    sc_out<T> out[8];
5    T buf[8];
6    void do_it() {
7      for(int i=0; i<8; i++) buf[i]=in[i];
8      for(int i=0; i<8-1; i++ ) {
9        for(int j=0; j<(8-i)-1; j++ ) {
10         if( buf[j]>buf[j+1] ) {
11           T tmp;
12           tmp = buf[j];
13           buf[j] = buf[j+1];
14           buf[j+1] = tmp;
15         }
16       }
17     }
18     for(int i=0; i<8; i++) out[i]=buf[i];
19   }
20   SC_CTOR( bubble ) {
21     SC_METHOD(do_it); sensitive << in;
22   }
23 };
```

Fig. 4. Bubble sort module

TABLE II

RESULTS FOR DIFFERENT INPUT SIZES OF MODULE `bubble`

Bit size	Clauses	Literals	Time(sec)
4	6390	14458	17.18
8	12754	28894	286.93
16	25482	57766	125.25
32	50938	115510	560.48

needed is shown. Due to the heuristic nature of the SAT solver the proof time might slightly vary as can be seen in case of bit size 8. But in general the run time needed increases with the bit size and is moderate even for larger bit sizes.

In a third series of experiments a scalable arbiter circuit has been studied that is frequently used in formal hardware verification (see e.g. [5]). This example has already been considered for SystemC designs in [8], where properties for *mutual exclusion*, *liveness* and *conservativeness* have been studied. While *mutual exclusion* and *conservativeness* could easily be solved based on BDD-based property checking in [8], the approach failed on larger designs for the property *liveness*, since in this case a large number of time frames has to be considered[2]. The new approach was also applied to all three properties and could clearly outperform the earlier technique. Due to page limitation only the results for the hardest instance are shown. In Table III the results are given for increasing number of arbiter cells (column *Cells*). In the second column the run times for the approach from [8] are given. The next two columns provide information on the SAT instance. In the last column the overall CPU time needed is reported. As can be seen, the new approach can prove the properties very fast, even for large instances, while the previous approach can only be applied for less than 10 cells.

[2]The property *liveness* states, that each request has to be confirmed by an acknowledge within $2 \cdot n$ time frames, where n is the number of arbiter cells.

TABLE III

RESULTS FOR ARBITER AND PROPERTY *liveness*

Cells	BDD-PC	CheckSyC		
	Time(sec)	Clauses	Literals	Time(sec)
5	2.14	6808	15728	0.15
6	38.71	9899	22873	0.17
7	315.92	13566	31350	0.25
8	2667.76	17809	41159	0.30
9	-	22628	52300	0.37
10	-	28023	64773	0.49
20	-	113653	262763	2.11
50	-	716143	1655933	27.21

V. CONCLUSIONS

In this paper the efficient property checker CheckSyC has been presented. In contrast to previous approaches, CheckSyC supports a larger set of SystemC constructs and due to its high capacity can also be applied in cases where properties argue over large observation windows - as this is typically the case on the system level.

It is focus of current work to integrate word-level information in the proof process to speed up property checking.

REFERENCES

[1] S. Liao, S. Tjiang, and R. Gupta, "An efficient implementation of reactivity for modeling hardware in the scenic design environment," in *Design Automation Conf.*, 1997, pp. 70–75.

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

[3] R. Drechsler, *Advanced Formal Verification*. Kluwer Academic Publishers, 2004.

[4] R. Drechsler and S. Höreth, "Gatecomp: Equivalence checking of digital circuits in an industrial environment," in *Int'l Workshop on Boolean Problems*, 2002, pp. 195–200.

[5] 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*, 2001, pp. 742–748.

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

[7] R. Drechsler and D. Große, "Reachability analysis for formal verification of SystemC," in *EUROMICRO Symp. on Digital System Design*, 2002, pp. 337–340.

[8] D. Große and R. Drechsler, "Formal verification of LTL formulas for SystemC designs," in *IEEE International Symposium on Circuits and Systems*, 2003, pp. V:245–V:248.

[9] G. Fey, D. Große, T. Cassens, C. Genz, T. Warode, and R. Drechlser, "ParSyC: An efficient SystemC parser," in *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, 2004, pp. 148–154.

[10] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 1579. Springer Verlag, 1999, pp. 193–207.

[11] P. Johannsen and R. Drechsler, "Formal verification on register transfer level – utilizing high-level information for hardware verification," in *IFIP Int'l Conf. on VLSI*, 2001, pp. 127–132.

[12] J. Bormann and C. Spalinger, "Formale Verifikation für Nicht-Formalisten (Formal verification for non-formalists)," *Informationstechnik und Technische Informatik*, vol. 43, pp. 22–28, 2001.

[13] T. Parr, *Language Translation using PCCTS and C++: A Reference Guide*. Automata Publishing Co., 1997. [Online]. Available: citeseer.ist.psu.edu/parr97language.html

[14] Synopsys, *Describing Synthesizable RTL in SystemC^{TM}, Vers. 1.1*. Synopsys Inc., 2002, available at http://www.synopsys.com.

[15] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Design Automation Conf.*, 2001, pp. 530–535.