Checkers for SystemC Designs*

Daniel Große Rolf Drechsler

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

Abstract

Today's complex systems are modeled on a high level of abstraction. In this context, C/C++-based description languages, like SystemC, become very important. The modeling features of SystemC enable adequate levels of abstraction, hardware/software integration and fast executable specifications. Using the SystemC design methodology, a system is partitioned into hardware and software. Then the modules are refined down to the implementation. Besides efficient modeling, the correct functional behavior is very important. Already today up to 80% of the overall design costs are due to verification. As the complete system cannot be formally verified, checking of the functional behavior during operation has to be considered.

In this paper an approach is presented that allows to check temporal properties for a SystemC design not only during simulation, but also after fabrication in form of an on-line test. The method translates the properties into synthesizable SystemC instructions. By this, the properties can be checked like HDL assertions during simulation and after production since they can be synthesized together with the system. The proposed approach enables a concise circuit and system verification methodology.

1. Introduction

While classical design approaches based on VHDL or Verilog mainly allow for describing on the RT-level, modeling on higher levels of abstraction becomes more and more important. Since first reference designs are usually modeled in C or C++ there is a strong interest in C-like description languages which cover all levels of abstraction. A very promising candidate in this area is SystemC [11]. 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. The well-known concept of hierarchical descriptions of systems is transferred to SystemC by describing a module as a C++ class [9, 12, 14]. 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.

But with the increasing complexity of today's systems it is a challenging task to ensure the correct functional behavior. In the meantime it has been observed that verification becomes the major bottleneck of modern circuit and system designs, since up to 80% of the overall design costs are due to verification. As alternatives to classical simulation formal methods have been proposed [3]. In equivalence checking formal tools are state-of-the-art [5]. In case of SystemC, examples of simulation based techniques are [13, 6]. First formal approaches to check the behavior of a circuit description in SystemC have been reported in [4, 8]. While equivalence checking can be applied to large designs, property checking is limited to the block level. But besides the verification of the blocks in a large system their mutual communication has to be checked.

There are several approaches to system level verification which are based on assertions [7]. The key idea is to describe expected or unexpected behavior directly in the device under test. These conditions are checked dynamically during simulation. An approach to check temporal assertions for SystemC has been presented in [13]. There, the specified properties are translated to a special kind of finite state machines (AR-automata). These automata are then checked during the simulation run by algorithms, which have been integrated into the SystemC simulation kernel. In contrast in [2] a method has been proposed to synthesize properties for circuits into hardware checkers. Properties which have been specified for (formal) verification are directly mapped onto a very regular hardware layout.

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

Following the latter idea in this paper a method is presented which allows checking of temporal properties for circuits and systems described in SystemC not only during simulation. A property is translated into a synthesizable SystemC checker and embedded into the circuit description. This enables the evaluation of the properties during the simulation and after fabrication of the system. Of course, with this approach a property is not formally proven and only parts of the functionality are covered. But the proposed method is applicable to large circuits and systems and supports the checking of properties in form of an on-line test. This on-line test is applicable, even if formal approaches failed due to limited resources.

The remaining part of the paper is structured as follows: In Section 2 an overview about SystemC is given. Section 3 describes the concepts and translation of a temporal property into a SystemC checker. Experimental results demonstrating the advantages of the approach are given in Section 4. Section 5 concludes the paper and summarizes the results.

2. 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.

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. The output of a system can be textual, using C++ routines like `cout` for instance, or waveforms. As a C++ class library SystemC can easily be extended by using the facilities of C++.

The following section describes how a temporal property is translated into a SystemC checker.

3. SystemC Checker

3.1. 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 the notation of the property checker from Infineon Technologies AG (see e.g. [10, 1] 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` are offsets. If all assumptions hold, all commitments in the proof part have to hold as well.

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`.

Each property can be translated to SystemC constructs. The result is a SystemC checker which is based on shift registers and additional logic representing the relations of the specified signals. The SystemC constructs needed for generation of a checker from a property can be limited to synthesizable SystemC code.

In the next section the overall flow for the translation of a property is discussed.

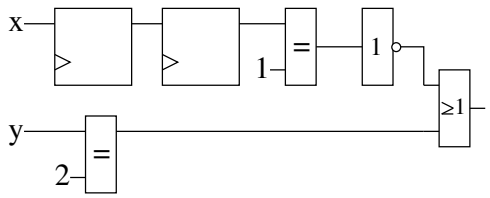

Figure 1. Shift register and logic for property `test`

3.2. Basic Idea of Checker Generation

At first the basic idea of the translation of a property into a checker is illustrated by the following example.

Example 2. *Consider again Example 1. For the property* `test` *it has to be checked that whenever signal* x *is* 1*, two time frames later* y *has to be* 2*. This is equivalent to* $\neg(x'' = 1) \vee (y = 2)$*, if* x'' *is* x *delayed by two clock cycles. If the equation evaluates to false the property is violated. Obviously the translation of the property can be expressed in SystemC. The basic idea of a hardware realization is shown in Figure 1. If the output of the OR gate is* 0 *the property fails.*

In general the translation of a property works as follows: Let P be a property which consists of the assumptions $A = (a_1, \ldots, a_m)$ and the commitments $C = (c_1, \ldots, c_n)$. Then the translation algorithm is based on four steps:

1. Parse P and determine the maximum offset o_{max} of the property by analyzing the time points of all a_i and c_j.

2. For each signal used in P generate a shift register of length o_{max}. Then the values of a signal at time points $t,\ t+1,\ \ldots,\ t+o_{max}$ are determined by the outputs of the flip-flops in the corresponding shift register. The offset i of a time point can directly be identified with the i-th flip-flop, if the flip-flops are enumerated in descending order. This is illustrated in Figure 2.

3. Combine the signals of each a_i (and c_j) as stated by the logic operations in its expression. Thereby the variables of the appropriate time points are used. In case of the interval operators `during` and `within` an AND and an OR of the resulting expressions is computed. The results of this step are the equations $\hat{a}_1,\ \ldots, \hat{a}_m,\ \hat{c}_1,\ \ldots, \hat{c}_n$ corresponding to the assumptions and commitments of P.

4. The final equation is $check_P = \neg \bigwedge_{i=1}^{m} \hat{a}_i \vee \bigwedge_{j=1}^{n} \hat{c}_j$.

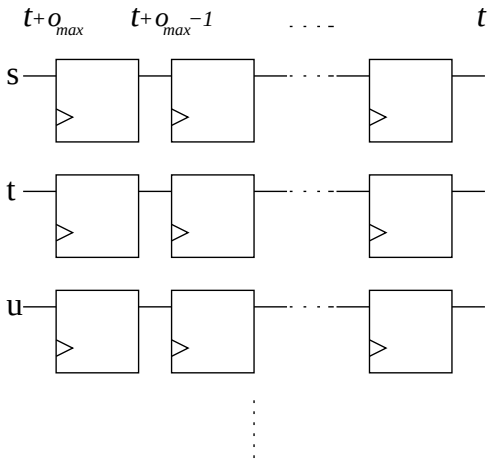

Figure 2. Mapping of time points

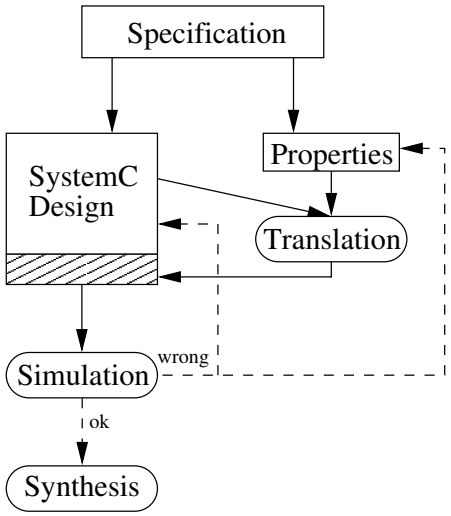

Figure 3. Work flow

Of course all described transformations from the property description into the resulting equation $check_P$ have to be performed by using SystemC constructs, i.e. the use of different data types and operators has to be incorporated. Finally, the property P can be checked by evaluating $check_P$ in each clock cycle during simulation or operation. In the next section some details about the transformation into SystemC code are given.

3.3. Transformation into SystemC Checkers

The work flow of the proposed approach is shown in Figure 3. At first the design has to be built and the specification has to be formalized into properties. Then the properties are translated to checkers and embedded into the design description (hatched area in the figure). If all checkers hold

```
template<class T >
class regT : public sc_module {
public:
    sc_in_clk clock;
    sc_in<T > in;
    sc_out<T > out;

    SC_CTOR(regT) {
        SC_METHOD(doit);
        sensitive_pos << clock;
    }
    void doit() {
        out = in.read();
    }
};
```

Figure 4. Generic register

```
regT<sc_int<8> > *r
 = new regT<sc_int<8> >("reg");
r->clock(clock);
r->in(a);
r->out(a_d);
```

Figure 5. Usage of generic register

during simulation the next step of the design flow can be entered.

A property is assumed to use only port variables and signals of a fixed SystemC module or from its sub-modules. During the translation for the variables of the properties shift registers have to be created as has been described in the previous section (Step 2). For this purpose a generic register as shown in Figure 4 has been modeled. The register delays an arbitrary data type for one clock cycle. If such a templated register is not directly supported by the synthesis tool, it is possible to replace every templated register with a version where the concrete input and output types are explicitly specified. The generic register can be used as shown in the example in Figure 5. There a register with an `sc_int<8>` input and output is declared and instantiated.

During the generation of the shift registers of length o_{max} for a variable, o_{max} generic registers have to be declared and instantiated. This is done in the constructor of the considered module. The necessary `sc_signals` (output variables of the registers) for the different time points are declared as member variables of the considered module. Their names are produced by adding the number of delays to the variable name. The absolute time points can not be used, because if a variable is employed in at least two properties the delay of the same time points may differ.

```
SC_MODULE(module) {
public:
    // ports
    sc_in_clk clock;
    ...

    // sc_signals for different
    // time points
    sc_signal<T> x_d1,x_d2;

    SC_CTOR(module) {
        // shift register
        regT<T> rx_d1 = ...
        rx_d1->clock(clock);
        rx_d1->in(x);
        rx_d1->out(x_d1);
        regT<T> rx_d2 = ...
        rx_d2->clock(clock);
        rx_d2->in(x_d1);
        rx_d2->out(x_d2);
        ...
    }
};
```

Figure 6. Insertion of a shift register for property `test`

Example 3. *Consider again Example 1. Let the data type of* `x` *be* `T`*. Let the property* `test` *be written for the SystemC module* `module`*. As has been explained above* `x` *has to be delayed two times. Then the resulting shift register is inserted into the module as shown in Figure 6.*

As can be seen in Figure 6 the data type of a variable used in a property has to be known for declaration of the `sc_signals` and shift registers. Thus, with a simple parser the considered SystemC module is scanned for the data types of the property variables.

The resulting code to check a property (equivalent to the equation $check_P$) is embedded into an `SC_METHOD` process of the module, which is sensitive to the module clock, i.e. the process is triggered every clock cycle. In the final step of writing SystemC code for the translated property the following is taken into account:

- The shift register for each variable used in a property is shared between different checkers.
- In case of an array access it has to be distinguished between an access to an array of ports and an access to a port which contains an array type. An array of ports is mapped onto different variables each representing an

```
// theorem: test
bool check_test = !( ( x_d2.read() == 1 ) ) | ( y.read() == 2 ) ;
if (check_test == false) {
    cout<<"@"<<sc_simulation_time()<<": THEOREM test FAILS!"<<endl;
}
```

Figure 7. Checker for property `test`

according index of the array. Furthermore the access operator `[]` has to be replaced accordingly.

- The operators of the property language have to be mapped onto its counterparts in C++, e.g. = to ==.
- The resulting checker formula is assigned to a Boolean variable `check_<property name>`. If this variable is false during simulation the property is violated and an according output is given using the `cout` routine. For the synthesis part an output port for the considered module has to be generated, which assumes zero if the property fails.

Example 4. *In Figure 7 the translated equation* `check_test` *for the property* `test` *is shown. If the property fails, this is prompted directly to the designer.*

3.4. Optimizations

All shift registers for different properties of one concrete module which are driven by the same clock, can be integrated into one clocked process. Then in the constructor `SC_CTOR` of the module instead of the shift registers only one clocked process has to be declared. In this process the according output variables are written, e.g. in case of the property `test` the process statements are:

```
x_d1.write(x); x_d2.write(x_d1);
```

So the number of `SC_METHODs` is reduced and the simulation speed increases (see also Section 4.2).

As has been explained in the previous section if the checkers are synthesized one-to-one for each property an output port is generated, which assumes the value zero if the property fails. This leads to a trade-off between good diagnosis and number of output pins. Diagnosis is easy if each property directly corresponds to an output pin, while many outputs require more chip area.

4. Experimental Results

The technique described above is experimentally studied. For this task a bus architecture has been modeled. In Figure 8 a block diagram of the bus architecture is shown.

The bus is described as a SystemC module, and masters and slaves can connect to the bus. The bus is divided into

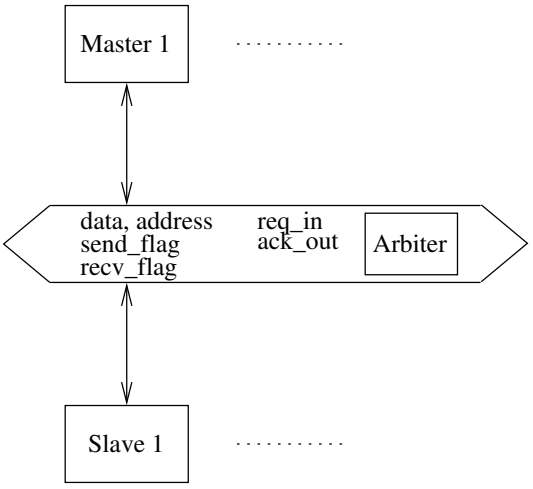

Figure 8. Bus architecture

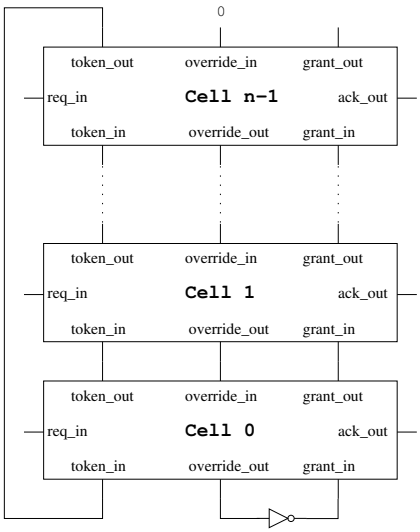

Figure 9. The integrated arbiter

a data part, an address part and a flag part. These are all `sc_inout`-ports and they have the type `sc_uint` with a scalable size to allow for variable data width, number of slaves, and number of masters. The address is used by the masters to address a slave. The flags `send_flag` and `recv_flag` are set during a bus transaction (see below).

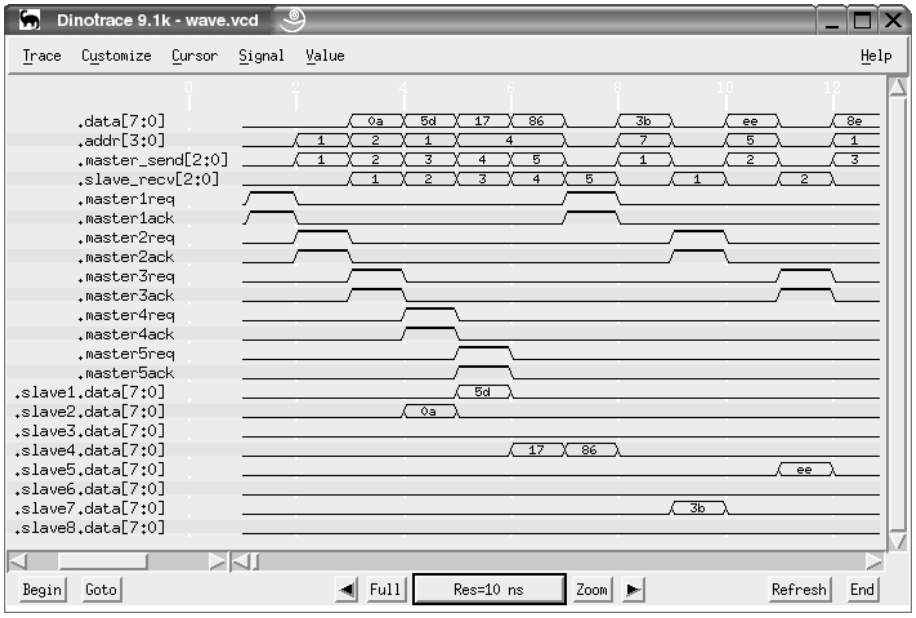

Figure 10. Simulation trace of a small bus example

Furthermore the bus contains a scalable arbiter. Thus the bus also has a request input and an acknowledge output for each master. The arbiter consists of n cells (one for each master) and combines priority arbitration with a round robin scheme. This guarantees that every master will finally get access to the bus. In Figure 9 the arbiter is shown. Summarized, the features of the bus are:

- Only masters can write to the bus and each master has a unique id.
- A slave has a unique address. This address is given at instantiation of the slave.
- A bus transaction works as follows:
  1. A master requests the bus via its request output. If access is granted see Step 2, otherwise the master waits for an acknowledgment.
  2. The master writes the target address and the data to the bus. Furthermore, the master writes its id to the `send_flag`. Then the master waits for an acknowledge that the slave has received the data via the `recv_flag` (id of the master at the `recv_flag`).
  3. If a slave detects its address on the bus, the slave reads the data and writes the id from the `send_flag` to the `recv_flag` of the bus.
  4. If the master detects its id on the bus, the data transmission was successful.

A wave form example of a bus with five masters and eight slaves is shown in Figure 10.

4.1. Checkers

In the following an informal description of the properties is given, which have been embedded as checkers into the bus module:

1. Two output signals of the arbiter can never become 1 at the same time (*mutual exclusion*).
2. The acknowledge is only set if there has been a request (*conservativeness*).
3. Each request is confirmed by an acknowledge within $2 \cdot n$ time frames (*liveness*).
4. If the bus has been granted for a master, the master writes its id to `send_flag` in the next cycle (*master id*).
5. If a slave has been addressed, the slave writes the master id (available at the `send_flag`) to the `recv_flag` (*acknowledge master*).

4.2. Simulation Results

All experiments have been carried out on an Intel Pentium IV 3GHz with 1GB RAM running Linux. Checkers

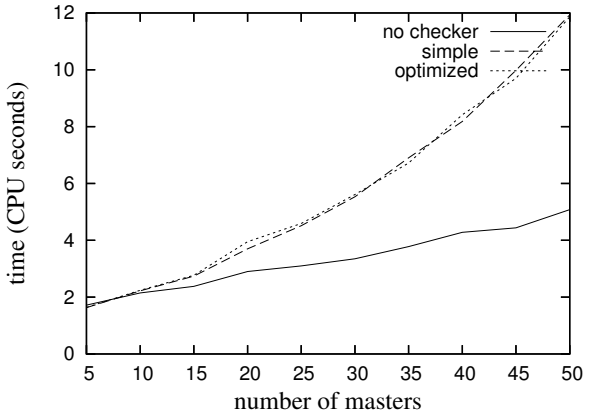

Figure 11. Comparison of simulation performance for checker *mutual exclusion*

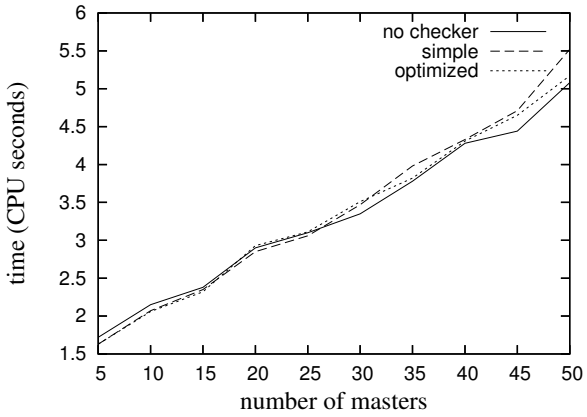

Figure 12. Comparison of simulation performance for checker *conservativeness*

have been generated for all described properties (see previous section). In the following for each property the simulation performance in case of no checkers, the simple approach, and the optimized approach are compared. For this task the bus model has been simulated for 100,000 clock cycles for a various number of masters. Note that the number of masters connected to the bus is equal to the number of arbiter cells. For the checkers described above we obtained the following results:

1. In Figure 11 the performance comparison for the checker *mutual exclusion* is shown. As can be seen the simulation time for the simple and the optimized approach increases with the number of masters. Both approaches behave similar since the observation window of the mutual exclusion property is 0, so no registers have to be created. For this reason no optimization is possible. The total runtime overhead is moderate, i.e. within a factor of two for 40 cells.

2. The simulation performance with and without the checkers for the *conservativeness* properties is nearly identical (see Figure 12). This is an expected behavior, because each conservativeness property only argues over two signals of each arbiter cell.

3. In Figure 13 the results for the *liveness* checkers are shown. The figure shows that the optimized approach leads to better results than the simple approach. Since the observation window of the liveness property is $2 \cdot n$ (where n is the number of masters) the number of `SC_METHODS` has been reduced effectively by optimization. However the runtime overhead compared to pure simulation is due to the significantly increasing size of the observation windows of the liveness properties.

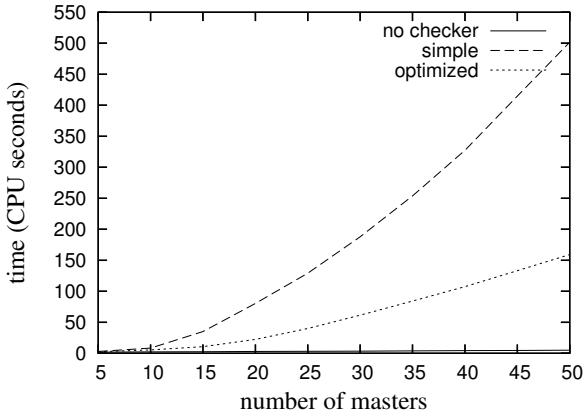

Figure 13. Comparison of simulation performance for checker *liveness*

4. The results for the checkers of *master id* show that there is a small benefit of the optimized approach over the simple approach (see Figure 14). In total these properties can be checked during simulation very fast.

5. As expected the *acknowledge master* property leads to the same performance as pure simulation, because this property could be described very compact. Figure 15 shows the diagram.

The experiments demonstrate that the overhead during simulation for properties with large observation windows is moderate, and negligible for properties with smaller observation windows.

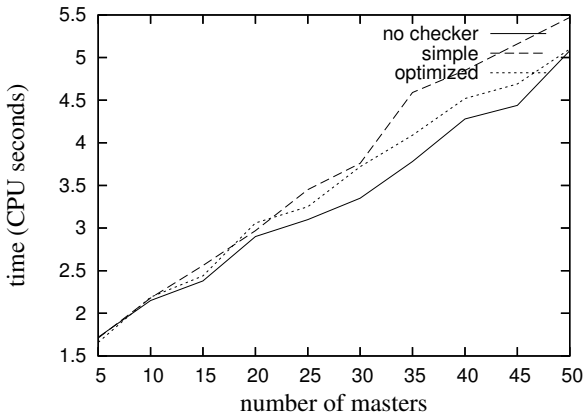

Figure 14. Comparison of simulation performance for checker *master id*

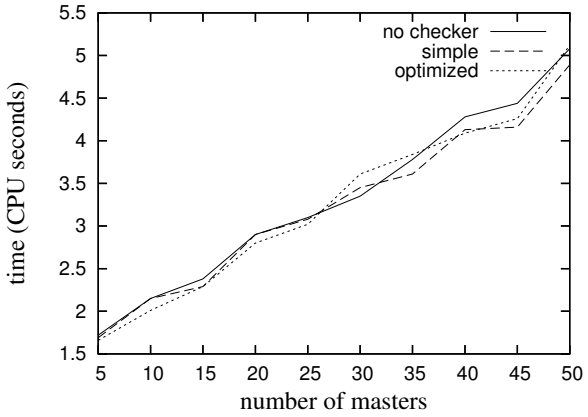

Figure 15. Comparison of simulation performance for checker *acknowledge master*

5. Conclusions

A new approach has been presented that allows to check temporal properties for SystemC designs not only during simulation but also after fabrication. Therefore each property is translated into a checker. Such a checker is based on synthesizable SystemC constructs and embedded into the design description. Experimental results have demonstrated the quality of the approach. In case of large observation windows of the checkers the runtime overhead is moderate, whereas for small observation windows it is negligible.

References

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

[2] R. Drechsler. Synthesizing checkers for on-line verification of system-on-chip designs. In *IEEE International Symposium on Circuits and Systems*, pages IV:748–IV:751, 2003.

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

[4] R. Drechsler and D. Große. Reachability analysis for formal verification of SystemC. In *EUROMICRO*, pages 337–340, 2002.

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

[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] H. Foster, A. Krolnik, and D. Lacey. *Assertion-Based Design*. Kluwer Academic Publishers, 2003.

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

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

[10] 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*, pages 127–132, 2001.

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

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

[13] 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.

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