Simulation-Guided Property Checking Based on Multi-Valued AR-Automata *

Jürgen Ruf, Dirk W. Hoffmann, Thomas Kropf, and Wolfgang Rosenstiel

Institute of Computer Engineering, University of Tübingen, Germany
{ruf,hoff,kropf,rosen}@informatik.uni-tuebingen.de

Abstract

The verification of digital designs, i.e., hardware or embedded hardware/software systems, is an important task in the design process. Often more than 70% of the development time is spent for locating and correcting errors in the design. Therefore, many techniques have been proposed to support the debugging process. Recently, simulation and test methods have been accompanied by formal methods such as equivalence checking and property checking. However, their industrial applicability is currently restricted to small or medium sized designs or to a specific phase in the design cycle. In this paper, we present a method for verifying temporal properties of systems described in an executable description language. Our method allows the user to specify properties about the system in finite linear time temporal logic (FLTL). These properties are translated to a special kind of finite state machines which are then efficiently checked on-the-fly during each simulation run. Properties may be placed anywhere in the system description and violations are immediately indicated to the designer.

1. Introduction

Assuring correctness of digital designs is one of the major tasks in the system design flow. Systems in our context are hardware systems or embedded hardware/software systems such as bus arbiters, automotive controllers or microprocessors. These systems are reactive, i.e., they are embedded in an interactive environment and have to react within certain time bounds.

The system design starts with an abstract model describing the main functionality. This model is then successively refined until a net-list is created that fulfills all system requirements such as timing and area constraints, power consumption, etc. Elimination of design errors can become very expensive, i.e., if errors are encountered in a later design stage. Hence, it is extremely important to find errors in early design stages.

State of the art validation techniques are simulation and test methods. Recently, formal methods such as equivalence checking [2] and model checking [3] found entrance into design laboratories. Model checking is an automated method working well on small or medium sized designs and is usually applied very early in the design process (RT level or higher). In contrast to this approach, equivalence checking has successfully been used for verifying large designs and is the matter of choice once a design is brought down to the gate level.

¿From a validation point of view, formal methods have the desired property that one single verification run implicitly covers 100% of all test cases. The major drawback of formal verification methods, however, is the limited size of verifiable systems. In contrast to formal verification methods, simulation based approaches only provide a partial test case coverage (defined by the simulated test-cases), but do not suffer from combinational explosion and can therefore be applied to very large systems.

Our approach aims at the verification of large systems described in executable system description languages. Therefore, we have chosen a simulation based approach for validating temporal properties of systems at all levels of abstraction. Our method takes simulation-runs generated during the validation phase and checks each run against one or more temporal specifications. We call this approach on-the-fly since the property checking algorithm is directly linked to the simulator and works like an observer during simulation.

To specify temporal properties, we introduce FLTL (finite linear temporal logic), a variant of linear time temporal logic (LTL [5]) which interprets formulas over finite traces. In contrast, the semantics of standard LTL is based on infinite runs. In addition, we support quantitative timed operators, e.g., to express that a certain event happens within a specified time interval. These properties are especially important for dealing with reactive systems. The temporal

*This work is supported by the German Research Grant (DFG) Project GRASP and the ESPRIT LTR Project 26241 (Prosper)

assertions can appear at any position in the system description and therefore be used like a standard assert-statement in languages like VHDL or C. Assert statements, however, can only check Boolean properties about the current state of a system while temporal properties are able to express properties about future states (e.g., liveness properties or reaction time).

The novelties of our approach can be summarized as follows: FLTL formulas are compiled to a special kind of finite state machines (called AR-automata) prior to simulation. This approach moves most of the computational complexity into the preprocessing phase and minimizes the computation overhead during simulation. We have chosen this approach as simulation speed is more important for our applications than memory consumption. In [11] an approach using active formula objects has been presented. These objects are created dynamically and checked until the formula becomes true or false. This approach has the advantage that no preprocessing step is required. However, it consumes a lot of memory and, more important, it can considerably slow down the simulator. Based on the feedback from industry, speed is the most important requirement imposed on simulation based verification algorithms. Therefore, the main motivation for our new approach was to develop a simulation-based verification method that minimizes the computation overhead during simulation, even for the price of higher memory consumption.

Compiled formulas are stored in an automata-database. Hence, an FLTL formula has to be compiled only once, and the created automata will automatically be reused in every new simulation run. The database-approach dramatically reduces the precomputation overhead.

Temporal asserts can be placed everywhere inside the HDL code with a special assert statement. This makes our approach easy to use and temporal specifications implicitly document the HDL code.

We do not make any restriction how temporal assertions can be nested. This is in contrast to [9] which only support a restricted temporal logic.

The temporal checker is linked into the executable specification. This enables full control over the simulation kernel. For example, the simulator can be stopped exactly at the cycle where a violation of a temporal specification has been detected. Moreover, violations can immediately be prompted to the user. This is not possible with loosely coupled systems that utilize a pipe-based communication with the simulation kernel.

This paper is organized as follows: Section 2 discusses the state-of-the-art in simulation-based verification. In Section 3 we define syntax and semantics of the the temporal logic FLTL (Finite Linear Temporal Logic). Section 4, we introduce accept/reject-automata (AR-automata) which are utilized to check an FLTL specification against a finite simulation run. The construction of AR-automata is presented in Section 5. In Section 6, we show how the state-space of AR-automata can be reduced considerably which enables the verification of complex temporal formulas. Section 7 outlines the integration of our approach into an industrial design language (SystemC). We conclude the paper with the presentation of some experimental results in Section 8 and a summary in Section 9.

2. Related Work

Several approaches have been proposed in the literature for checking temporal specifications during simulation. The technique presented in [1] is utilized for checking event-patterns in VHDL descriptions. Similar to our approach, the patterns are directly annotated as special comments inside the hardware description language (VHDL). The patters may be clustered hierarchically, i.e., a pattern may contain sub patterns which have to appear in the specified order. As the patterns can only be linearly chained, the supported logic is less expressive than the logic introduced in this paper.

Nelson and Jones describe in [9] a simulation-based checking algorithm based on a translation of the properties into finite state machines. The composition of the finite state machines is restricted to sequential chaining of two state machines. Hence, temporal operators can only be combined sequentially restricting the supported logic to temporal formulas that are not nested. Sequential chaining is expressed by a newly introduced “THEN” operator.

Canfield et. al. proposed a method based on formula manipulation [4]. This approach checks the boolean fraction in the current simulation cycle. If no violation is detected in the current cycle, the temporal operators are unrolled by their fix-point definition and the algorithm is repeated. The approach requires less memory than approaches based on finite-state-machines. However, the algorithm requires considerably more computation overhead in each simulation cycle.

In contrast to compilation based approaches which consume the same computation time independent of the formula size, the computation overhead of the algorithm in [4] increases with the size of the formula to check. The approach is, however, well suited in scenarios where memory consumption is more important than simulation time.

Verification techniques have also been combined with test-bench generation methods and realized in commercially available tools (e.g., Specman Elite[1] and Vera[2]). Both tools provide object-oriented languages enriched with special constructs for specifying temporal behavior. Temporal

[1] www.verisity.com
[2] www.synopsys.com

specifications are therefore part of the test-bench in contrast to our approach where temporal formulas can be placed inside the HDL code itself. Furthermore, we have integrated the checker inside the executable specification, i.e. the checker is realized as a linkable library. This implies that no additional tools are necessary for using the checker and the user need not learn a new language.

3. Linear Temporal Logic

3.1. Syntax

For the rest of this paper, assume $Vars = \{a, b, c, \dots\}$ is a finite set of distinct symbols, called the *variable domain*.

Definition 1 *A trace $T[n..m]$ $(m \geq n)$ is a mapping $T : \{n, \dots, m\} \to 2^{\text{Vars}}$. If n and m are clear from the context, we often simply write T instead of $T[n..m]$. The set of all traces is denoted by $\mathcal{T}$. The set of all traces $T[0,m]$ with $m = \infty$ is denoted by $\mathcal{T}^\infty$.*

Definition 2 *Let $T[0,m], T'[0,n]$ be two traces with $n > m$. T' is called a* trace extension *of T iff*

$$\text{for all } j \text{ with } 0 \leq j \leq m : T(j) = T'(j) \qquad (1)$$

Definition 3 LTL*, the set of all Linear Temporal Logic formulas, is the smallest set satisfying*

$$\begin{array}{l} \text{Vars} \subset \text{LTL}, \textit{ and} \\ \neg f, f \wedge g, X_{[m]}f, G_{[m,n]}f, F_{[m,n]}f \in \text{LTL} \\ \textit{iff}\ f, g \in \text{LTL} \textit{ and } m \in \mathbb{N} \textit{ and } n \in \mathbb{N} \cup \{\infty\} \end{array}$$

3.2. Semantics

LTL formulas are interpreted over traces.

Definition 4 *The satisfiability relation $\models_i \subset (\mathcal{T}^\infty, \text{LTL})$ is defined recursively over the structure of LTL formulas:*

$$\begin{array}{lll} T \models_i a & \textit{if} & a \in T(i) \\ T \models_i \neg f & \textit{if} & T \neg \models_i f \\ T \models_i f \wedge g & \textit{if} & T \models_i f \textit{ and } T \models_i g \\ T \models_i X_{[m]}f & \textit{if} & T \models_{i+m} f \\ T \models_i G_{[m,n]}f & \textit{if} & \textit{for all } j \textit{ with } i+m \leq j \leq i+n \\ & & \textit{holds that } T \models_j f \\ T \models_i F_{[m,n]}f & \textit{if} & \textit{there ex. a } j \textit{ with } i+m \leq j \leq i+n \\ & & \textit{such that } T \models_j f \end{array}$$

The standard temporal operators (F,G) are special cases of the timed operators by instantiating m, n with 0 and ∞, respectively.

We now define the semantics of LTL in terms of a satisfiability relation.

Definition 5 *Let f be a LTL formula and $T \in \mathcal{T}^\infty$ be a trace. T is called to* satisfy *f (written as $T \models f$) iff*

$$T \models_0 f. \qquad (2)$$

3.3. Interpreting LTL Formulas over Finite Traces

We now interpret LTL formulas over finite traces. We call the resulting logic *Finite Linear Temporal Logic (FLTL).*

Definition 6 *Let $T[0..n]$ be a trace and f be a LTL formula. f is called* true *with respect to T (denoted by $T \models f$) if for all trace extension $T'[0..\infty]$ of T holds that $T' \models f$. f is called* false *with respect to T if there exists no trace extension $T'[0..\infty]$ of T such that $T' \models f$. Otherwise f is called* pending.

4. AR-automaton

Definition 7 *An AR-automaton is a 5-tuple $\mathcal{A} = (S, \to, A, R, s_0)$ where $S = \{s_1, \dots, s_n\}$ is a finite set of states, $\to$ is the transition relation, $A \subset S$ is the set of* accepting states*, $R \subset S$ is the set of* rejecting states*, and $s_0 \in S$ is the start state of A. The input of A are all elements of 2^{Vars}.*

We write $s_i \xrightarrow{a} s_j$ to express that there is a transition from s_i to s_j labeled with a.

Definition 8 *Let $\mathcal{A}$ be an AR-automaton and $T[0..m]$ be a trace. A* run *of T with respect to $\mathcal{A}$ is a sequence of states $s_0, s_1, \dots, s_n$ such that $s_i \xrightarrow{T[i]} s_{i+1}$ holds for $0 \leq i < m$.*

Note that a single trace $T[0..m]$ may induce several runs as the automaton can be nondeterministic.

Definition 9 *Let $\mathcal{A} = (S, \to, A, R, s_0)$ be a deterministic AR-automaton and $T[0..m]$ be a trace.*

- *T is called an* accepted trace *if for the run $s_0, s_1, \dots, s_{m+1}$ induced by T, there is a j with $0 \leq j \leq m+1$ with $s_j \in A$ and for all $k < j$ holds $s_k \notin R$. Accordingly, this particular run is called an* accepted run.
- *T is called a* rejected trace *iff for the run $s_0, s_1, \dots, s_{m+1}$ induced by T, there is a j with $0 \leq j \leq m+1$ with $s_j \in R$ and for all $k < j$ holds $s_k \notin A$. Accordingly, this particular run is called a* rejected run.

5. Constructing AR-automata

This section addresses the construction of an AR-automaton for a given FLTL formula. The atomic AR-automaton shown in Fig.1 accepts a trace if a signal is true in the current simulation cycle. The doubly circled state represents the initial states. The states labeled with “A” belong to the set A of *accepting* states and the states labeled with R belong to the set R of *rejecting* states. For the rest of this paper, we do not distinguish between the state set representation A and the labeling representation “A”.

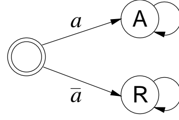

Figure 1. the basic AR-automaton

Table 1. Inheritance patterns

strong	weak
a state will be labeled with "A" if all sub-states are labeled with "A"	a state will be labeled with "A" if one sub-state is labeled with "A"
a state will be labeled with "R" if one sub-state is labeled with "R"	a state will be labeled with "R" if all sub-states are labeled with "R"

We now describe the main manipulation operations needed for the AR-automaton construction of nested FLTL formulas. The key idea is to build the AR-automaton for a given FLTL formula in a bottom-up manner starting with atomic signals. The constructed AR-automaton are then consecutively composed to more complex AR-automaton with respect to the operators in the FLTL formula.

The swap operation exchanges the sets A and R, i.e., a state labeled with "A" will be labeled with "R" and vice versa. This operation corresponds to the negation in the FLTL formula.

A major operation during the bottom-up construction is to remove nondeterminism, i.e., the translation of a nondeterministic to a deterministic state machine. With respect to removing nondeterminism, AR-automaton can be treated like standard finite state machines enabling the application of standard algorithms (e.g., sub-set construction [10]). We now discuss how the A-set and R-set of the new AR-automaton are inherited. Applying the standard technique for determining an AR-automaton, we obtain a new automaton whose states are sets of original states as shown in Figure 2 (the numbers below the states represent unique state identifiers and show the construction of new states by sets of original states). For constructing the A-set and the R-set of the new automaton, we distinguish two *inheritance patterns* as shown in Table 1. The type of inheritance which is used in the current construction step is determined by the temporal operator in the FLTL formula (see below).

A third operation is the *merge* operation. This operation takes two AR-automaton $\mathcal{A}$ and $\mathcal{B}$ and constructs one new AR-automaton by joining the initial states and appending the remaining transition relations of both AR-automata. This operation is used to build a new AR-automata for the conjunction respective disjunction. The left AR-automaton in Figure 2 shows the merged AR-automata for the signals a and b.

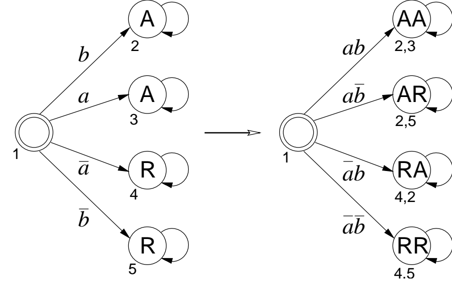

Figure 2. left: original AR-automaton, right: deterministic AR-automaton

Whether two merged AR-automaton are conjunctively or disjunctively connected depends on the inheritance pattern that is used for constructing the new A-set and B-set. Strong inheritance leads to conjunction and weak inheritance to disjunction.

The operation "append init loop" adds new transitions all starting from the initial state and leading to the initial state for all possible variable combinations. This operation is exemplarily shown in figure 3. This operation is used to build an AR-automaton for the globally (using strong inheritance while removing nondeterminisms) resp. eventually operator (using weak inheritance while removing nondeterminisms) with an infinite time bound.

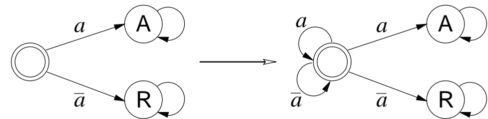

Figure 3. left: original AR-automaton, right: added initial loop

Adding a chain of new initial states and connecting all of them with the initial state of the original operator results in the construction of the globally resp. eventually operator with lower and upper time bounds. This operation is shown in Figure 4. In the case of an operator with interval $[a,b]$ we have to add b new states. The first $b-a+1$ states of the chain bocome the new initial states. In the case of $a=0$, the initial state of the original AR-automaton remains initial. The next time operator is a special case of this operation where $a=b$, i.e. we have to add a chain of length a and

only the first state becomes the new initial state.

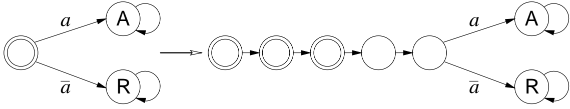

Figure 4. left: original AR-automaton, right: added chain [2,4]

In order to obtain correct AR-automata, we have to remove interminism after each operation.

The formal correctness proof of the construction techniques is skipped due to space limitations.

6. Reducing the State Space

Naive application of the construction rules can easily lead to huge AR-automata. This is in particular true for deeply nested formulas or formulas containing temporal quantifiers with very large time bounds. Looking closer into the structure of the generated AR-automata, however, one can notice that the number of equivalent states becomes very large. In other words, many states in the constructed AR-automata are bisimilar [8].

The standard method for merging bisimilar states is based on a partitioning algorithm [8]. It starts with a coarse partition and consecutively applies refinement steps. The algorithm is then repeated until a fix-point is reached. In our application, we utilize the standard partioning algorithm by initially defining three equivalence classes:

1. all accepting states
2. all rejecting states
3. the states which are neither accepting nor rejecting

The reduction is exemplarily shown in Figure 5.

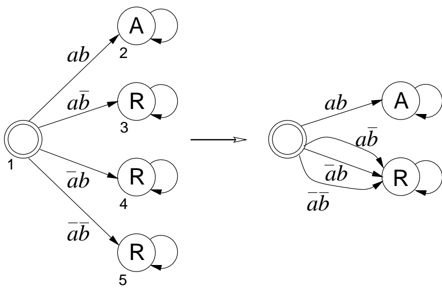

Figure 5. left: original AR-automaton, right: reduced AR-automaton

As will be demonstrated in the experimental results section, the bisimulation reduction dramatically shrinks the state-space of AR-automata. To avoid an early combinational explosion of the state-space, the bisimulation reduction is applied after each determinization step.

A second reduction technique for AR-automaton is called *pruning*. In constrast to bisimulation, this technique is very fast, but the reduction might be weaker than bisimulation as it is an approximation of the bisimulation classes. In a first step we build a class by joining all accepting states and a second class by joining all rejecting states. All other states build their own equivalence class. Then we collect all successors of accepting resp. rejecting states and add them to the corresponding equivalence class.

In the example shown in Figure 5, pruning produced the same AR-automaton as the bisimulation reduction, but the experimental results demonstrate that this is not necessarily true.

7. Implementation using SystemC

For our implementation of the simulation-based property checker we have chosen the SystemC language [12]. SystemC allows the specification of systems on various levels of abstraction and it provides fast simulation speed due to compiled executable specifications (simulation kernel + system description). Moreover, SystemC is open source and therefore easily extendable.

We have realized the temporal checker as a stand-alone library which is linked to the simulation kernel. Since the checker is realized as a standard SystemC process, it is running in parallel to the simulation. This implies that property violations can be prompted immediately to the user. Moreover, temporal assertions may be placed everywhere in the system description.

The user adds properties with the `sc_assert` command. `sc_assert` takes a FLTL formula represented by a string-argument and synthezises the corresponding AR-automaton. The command adds the generated AR-automaton to the checker process which is then responsible for stepping forward in the AR-automaton corresponding to the actual simulation.

The formulas may include arbitrary C++-functions (returning boolean values) instead of a Boolean signal. These functions are evaluated in each simulation cycle and the function result will be treated as the value of a Boolean signal.

The AR-automaton synthesis is realized in Java. The generated AR-automata are stored in a database. Whenever an FLTL formula is going to be translated it is first checked wether an adaption of the formula, i.e., a formula which can be obtained by renaming the signals of an already translated formula, is already stored in the database. Hence, the database is functioning as a global cache for AR-automaton-objects and dramatically reduced the time

needed for preprocessing. The database does *not* influence the time spend during simulation. Figure 6 gives an overview of our implementation.

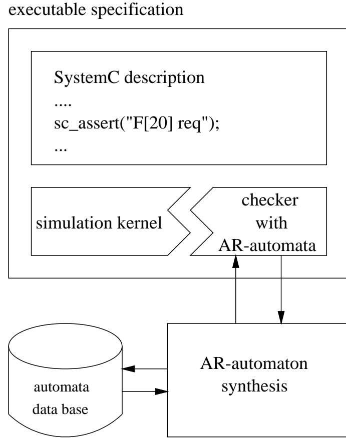

Figure 6. Architecture of our implementation

8. Experimental Results

In this section, we present experimental results obtained by applying our method to a scalable bus arbiter. The arbiter is an often used benchmark in the area of formal methods [7, 6]. The arbiter combines a priority arbitration with a round robin technique for guaranteeing fairness, i.e., each requesting cell will finally get access to the bus. A request is implemented on the hardware side by a persisting signal (req). The arbiter has the same number of cells as there are requesting signals.

For verification, we have randomly generated persisting request signals. We have checked the following formulas on-the-fly:

- Mutual exclusion: $G(OneHot)$

  $OneHot$ is a C++-function checking that maximal one acknowledge signal is high. As described in the implementation section, this function will be evaluated in each simulation cycle and the result "false" will be prompted to the user and the simulation can be stopped.

  The property guarantees that no two acknowledges appear simultaneously.

- Reactivity: $F[2n]ack_i$

  n is the number of arbiter cells. We place this property in the system description where the signal req_i is set to true. This means that the `sc_assert` command will introduce this formula whenever the ack_i signal raises. This property checks that every request is followed by an acknowledge within $2n$ time steps.

- Conservativenes: $G(ack_i \rightarrow req_i)$

  This formula is introduced before the simulation is invoked, i.e. it will be globally checked.

  No acknowledge is granted without a request.

The graph in Figure 7 shows the runtime overhead necessary for checking the described properties (for n cells, $2n+1$ properties have to be checked simulanously). The graph visualizes the overhead in function of the system size (measured in the number of arbiter cells) and the number of simulation cycles. The measured overhead ranges between 5% and 30%. It decreases with the number of simulation cycles since the time-overhead for reading the AR-automata from the database is constant (for one fixed system size), but simulation time and checking time decrease. This example is a worst-case scenario, since the specified properties fully describe the functional behavior of the system, but in real-world examples, the systems are larger and only a part of the functional behavior has to be represented by the properies (e.g. critical time bounds or critical safety assertuions). In comparison to this approach, the method presented in [11] takes two times the simulation time (i.e. 100% overhead) if the parallel property checking is activated.

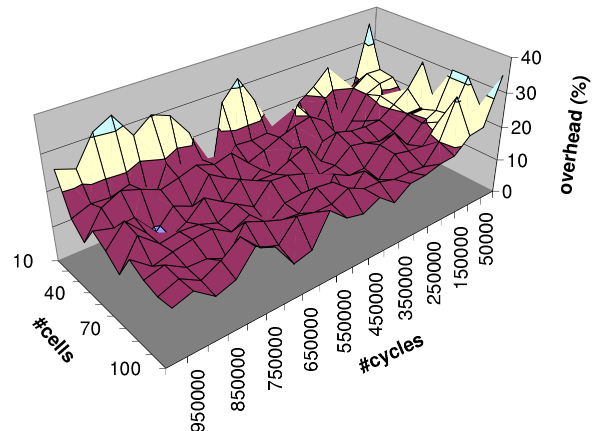

Figure 7. fraction of runtime using the checker to pure simulation time

In Table 2, the different state space reduction techniques are compared. The translated FLTL formula is G (F[n] s), i.e., in each time-frame of n cycles, signal s has to be high for at least one cycle.

If no reduction technique is used, the state-space grows exponentially. The exponential blow-up is completely

Table 2. AR-automata sizes in number of states

#cells	no red.	pruning	bisim.
2	1433	7	4
4	5241	11	6
6	11069	15	8
20	-	43	22
40	-	83	42
60	-	123	62
80	-	163	82
100	-	203	102
120	-	243	122
140	-	283	142
160	-	323	162
180	-	363	182
200	-	403	202

avoided, however, if one of the reduction methods is applied after each construction step. The bisimulation reduction decreases the number of states to 50% of the number of states obtained by pruning. Pruning, however, only requires 15% of the runtime of the bisimulation reduction.

9. Summary

We have presented a simulation-based approach for checking temporal assertions (FLTL formulas) in digital systems. FLTL formulas are compiled to AR-automata prior to simulation. Due to the preprocessing phase of the translation the computation-overhead shrinks to 5% to 30%. This is a promising result as simulation-speed is considered one of the main criterias for applying simulation-based property checking in real-life scenarios. To further decrease the preprocessing time, compiled formulas are stored in an automata database. AR-automaton of already compiled FLTL formulas are automatically retrieved from the database. Our approach supports nested temporal formulas and allows the user to place assertions anywhere inside the HDL code.

We have also presented a technique to efficiently reduce the state space of the generated AR-automaton. This technique approximates the bisimulation reduction and can dramatically decreases the number of states.

For testing our approach with an industrial strength system description language, we have integrated the algorithms into the SystemC simulation kernel. This integration allows a fast checking of simulation runs of systems on various levels of abstraction.

References

[1] L. Augustin, B. Gennart, Y. Huh, D. Luckham, and A. Stanculescu. Verification of VHDL designs using VAL. In *Design Automation Conference (DAC)*, pages 48–53. ACM/IEEE, 1988.

[2] D. Brand. Verification of Large Synthesized Designs. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 534–537, Santa Clara, California, November 1993. ACM/IEEE, IEEE Computer Society Press.

[3] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–33, Washington, D.C., June 1990. IEEE Computer Society Press.

[4] W. Canfield, E. Emerson, and A. Saha. Checking formal specifications under simulation. In *International Conference on Computer Design (ICCD '97)*. IEEE Computer Society Press, 1997.

[5] E. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072, Amsterdam, 1990. Elsevier Science Publishers.

[6] T. Kropf. *Formal Hardware Verification – Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*. Springer Verlag, state of the art report edition, August 1997.

[7] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[8] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1980.

[9] B. Nelson and R. Jones. Simulation event pattern checking with proto. In *SHDL 1994*, 1994.

[10] M. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:115–125, 1959.

[11] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation based validation of fltl formulas in executable system descriptions. In R. Seepold, editor, *Forum on Design Languages (FDL 2000)*, pages 311–319, Tübingen, Germany, September 2000. Sig.-VHDL and ECSI.

[12] Synopsys Inc., CoWare Inc., and Frontier Design Inc., www.systemc.org. *SystemC Version 1.0 User's Guide*.