TR 88.390 (H-0097) December 30, 2001 Computer Science

IBM Research Report

Design Exploration through Model Checking

Shoham Ben-David, Anna Gringauze, Sharon Keidar, Baruch Sterin, Yaron Wolfsthal
IBM Research Division
Haifa Research Laboratory
Israel

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Design Exploration through Model Checking

Shoham Ben-David, Anna Gringauze, Sharon Keidar, Baruch Sterin, Yaron Wolfsthal

IBM Research Laboratory in Haifa

Abstract. In recent years, the technique of symbolic model checking has proven itself to be extremely useful in the verification of hardware. However, after almost a decade, the use of model-checking techniques is still considered complicated, and is usually left for the experts. In this paper we address the question of how model-checking techniques can be made more accessible to the hardware designer community. We introduce the concept of *exploration* through model-checking, and demonstrate how, when differently tuned, the known techniques can be used to easily obtain interesting traces out of the model, rather than used for the discovery of hard-to-find bugs. We present a set of algorithms, which support the exploration flavor of model checking.

1 Introduction

The application of model checking in industrial settings requires a high level of user expertise to be able to withstand the state-space explosion problem. [2, 13, 10, 11]. The main reason is that many of the methods used for overcoming the size problem are not completely automated. Rather, these methods frequently draw on the insight and experience of the user for their success. User expertise is also required to model input behavior. In this activity, the user must work carefully to avoid false negative and false positive results when restricting input behavior to avoid state-space explosion. These application challenges, together with the need to master formal languages (in particular temporal languages), have established industrial model checking as a domain where a high level of expertise is required.

A significant number of methods have been proposed to withstand the state explosion problem [2, 3, 16, 17]. Some of these methods, however, required an even higher degree of expertise from users, as well as significant insights into the algorithmic nature of these methods. This has made model checking accessible only to trained verification engineers, thus limiting prospects for wide-scale deployment of model checking.

In this paper we take a different direction. Our investigation focuses on making model checking (and the associated benefits and impact) accessible to the non-expert user. Specifically, we aimed our efforts at reaching the community of *design engineers*, and providing them with a methodology and tools to develop and debug freshly-written HDL code; currently, no adequate cost-effective means exist for this purpose. Indeed, contemporary hardware design methodologies involve the creation of relatively small design blocks, which are only subjected to verification at the unit level (unit is an ensemble of blocks). Block level verification is typically tedious and costly, and is thus generally skipped or reduced to a minimum; this has a detrimental impact on time-to-market and overall design quality.

To address the above problems, we propose the paradigm of *Design Exploration* through model checking. This paradigm provides a means for the designer to explore, debug and gain insight into the behaviors of the design at a very early stage of the implementation—before verification has

even started. In this paradigm, the design engineer specifies a behavior of interest. The exploration tool then uses model checking techniques to find one or more execution sequences compliant with the specified behavior. When presented with such an execution sequence ("trace"), the designer is essentially furnished with an insight into the design behavior, and specifically with a concrete scenario in which the behavior of interest occurs. This scenario can then be closely inspected, refined, or abandoned in favor of another scenario. Using model checking for exploration provides two important advantages over traditional simulation. These are (1) the ability to specify scenarios of interest without specifying the inputs sequences required to reach them, and (2) the ability to reason about multiple executions in parallel, rather than one at a time.

The exploration paradigm presents some new challenges which were not raised in traditional model checking. First, as design exploration is geared for use by non-experts, it is important to hide the difficult parts of the technique, namely, the need to learn new languages and the need to accurately describe input behavior. Second, the model checker should be tuned to algorithmically support design exploration—in order for the new paradigm to be applicable and accepted, the underlying tool should quickly and easily provide as much information as possible to the design engineer exploring the hardware design. We present several algorithms which support the exploration paradigm. These include the generation of disjoint multiple traces, the production of maximal partial trace when no full trace exists, the interactive mode, where new requests can be made after all calculations ended, and the integration with a simulator.

Note that the size problem is not addressed in this paper. The struggle with size is minimized by restricting the application of exploration to small hardware models (which is consistent with the purpose of design exploration to serve as a block-level [1] design tool).

The problem of making model checking easier to access has been addressed before. Fisler in [12] and Amla et al in [1] discuss the usage of *timing diagrams* for specification, as those are a commonly used and visually appealing specification method for hardware designers. Winters and Hu [18] propose the approach of automatic source-level optimizations, to make models written by novices more efficiently processed by the model checker Murϕ. De Palma et al [9] have approached the usability problem by restricting the specification repertoire to a finite set of graphical, intuitive templates.

The rest of this paper is organized as follows: Section 2 describes the concept of design exploration. Section 3 presents the key algorithms required to support the design exploration paradigm. A summary of our experience with the deployment of an experimental exploration system is given in Section 4, and in Section 5 we conclude and point to directions for future research.

2 Design Exploration - Basic Principles

The exploration paradigm presents new challenges to the model checking techniques in terms of both ease-of-use and performance. In this section, we describe how the difficult parts of using model checking can be hidden from the design engineer engaged in exploration (herein abbreviated as "the user"). In section 2.1, we describe how the need for learning new languages can be avoided by using a simple graphical specification formalism, which provides a natural way for designers to state their intent. In section 2.2 we describe how control over input behavior can

[1] Blocks are small models of about 100 state variables

be easily achieved. In section 2.3 we link exploration with the known model-checking techniques, and outline the fundamental Design Exploration algorithm which relies on on-the-fly verification of RCTL formulas [5]. A set of new Model Checking algorithms which help implement design exploration are described in section 3.

2.1 The Visual Specification Formalism

In the Design Exploration paradigm, a design behavior is specified as a *path*, or a *sequence of events*. To specify a behavior of interest, the designer creates a graphical representation of an ordered sequence of events. For each event the user specifies the Boolean expression which defines the event. Thus, no temporal logic is required, nor are temporal formulas written to specify the behavior of interest.

As a simple example, consider a state machine "machine(0:3)", with 16 possible values. Suppose the designer is interested in seeing the state machine pass through states 4 and 6 and then reach state 1—not necessarily in consecutive cycles. The way it is expressed is by a graphical path description as shown in Figure 1 below. The translation of this path specification into a CTL

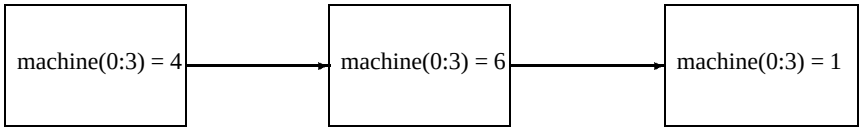

Fig. 1. Simple Path Specification

query [8] is given in formula 1 in section 2.3 below. This path specification drives the model checker to look for a trace with a state where machine(0:3) has the value of 4, then in a later state the value is 6, and on the final state of the trace, machine(0:3) has the value of 1. Although restrictive, we feel this path specification formalism is expressive enough to describe behaviors of interest. [2]

In order to find a compliant trace for the specified path, input behavior should also be supplied, as discussed in the next section.

2.2 Controlling Input Behavior

In the design exploration paradigm, the user should be able to produce first traces with minimal effort. Input signals should therefore have a default environment, to save the effort of assigning a behavior to each of them. We chose this default to be a "free" behavior, that is, a full non-deterministic behavior. Thus, even with no prior knowledge in formal methods, and with minimal effort, the user is able to generate initial traces. Generally, the first traces may exhibit illegal behavior due to the unrestricted behavior. The user then moves to restrict input behavior as

[2] We extended the notion of *events* to the more expressive notion of *phases*, which may last any number of cycles. However, this is beyond the scope of this paper.

needed. The more the user is willing to invest in this process the more accurate the input behavior will get.

In our experimental system described in section 4, the user is given a variety of ways to restrict input signal behavior. These include the ability to describe a deterministic behavior through a graphical editor, and the use of predefined state machines. Unless a very complicated input behavior is needed, in which case it should be modeled using HDL with non-deterministic extension, the user can easily define the desired behavior through graphical means.

2.3 The Mechanics of Design Exploration: The Link to Model Checking

In order to find a compliant trace to a given path specification, the design exploration algorithm translates the given path into an existential formula ϕ and send together with the model M to an underlying model checker. For example, the path specification given in Figure 1 is translated into CTL [8] as follows (For the sake of readability, we use the term ma instead of machine(0:3)):

$$E[ma \neq 4\, U\ ma = 4 \wedge EXE[ma \neq 6\, Uma = 6 \wedge EXE[ma \neq 1\, U\ ma = 1]]] \qquad (1)$$

Note that path specifications can be translated into RCTL formulas [5]. Since such formulas are very efficiently processed by on-the-fly model checking [5, 14], we base design exploration on the on-the-fly verification algorithm. When an RCTL formula ϕ is given, we use the algorithm given in [5] to translate it into a *Sugar* [6] regular expression. For example, formulas 1 is translated into *Sugar* as follows:

$$\{ma \neq 4[*], ma = 4, ma \neq 6[*], ma = 6,\ ma \neq 1[*], ma = 1\}(\textbf{false}) \qquad (2)$$

As described in [5] we automatically build out of ϕ an automaton A_ϕ, and a new $EF(p_\phi)$ type formula, such that

$$M \models \phi \iff M \times A_\phi \models EF(p_\phi)$$

where M is the model under test. The automaton built for formula 2 is given in Figure 2 below, written in the SMV [15] language. The automaton is accompanied by an $EF(p_\phi)$ type

```
VAR aut: { 0, 1, 2, 3, 4, 5, 6 };
ASSIGN
    init(aut) := { 1, 2 };
    next(aut) :=
    case
        aut = 1 ∧ ma ≠ 4 : { 2, 1 };
        aut = 2 ∧ ma = 4 : { 4, 3 };
        aut = 3 ∧ ma ≠ 6 : { 4, 3 };
        aut = 4 ∧ ma = 6 : { 6, 5 };
        aut = 5 ∧ ma ≠ 1 : { 6, 5 };
        1 : 0;
    esac;
```

Fig. 2. A Non-Deterministic Automaton in the SMV Language

specification, which is presented in Equation 3.

$$EF((aut = 6) \wedge (ma = 1)) \tag{3}$$

As mentioned above, this formula is then verified on-the-fly, while computing the reachable states space. The basic on-the-fly model-checking and trace generation algorithms are given below in Figure 3. We use the term *found* to indicate the BDD [7] representing p_ϕ. The on-the-fly algorithm saves the *new* sets of states, which are computed in every iteration of reachability analysis, as $S_0 \cdots S_n$ (line 4). This is done to make trace generation more efficient. These *new* sets are sometimes called *doughnuts*. In the sequel, we use the on-the-fly algorithm to present our enhanced algorithms for exploration.

```
1 reachable = new = initialStates;
2 i = 0;
3 while ((new ≠ ∅)&&(new ∩ found = ∅)) {
4       S_i = new;
5       i = i+1;
6       next = nextStateImage(new);
7       new = next \ reachable;
8       reachable = reachable ∪ next;
9 }
10 if (new = ∅) {
11     print "No trace exists for this path";
12     return;
13 }
14 k = i;
15 print "Trace found on cycle k";
16 good = new ∩ found;
17 while (i>=0) {
18     Tr_i = choose one state from good;
19     if (i>0) good=pred(Tr_i)∩S_{i-1};
20     i = i-1;
21 }
22 print "Trace is:" Tr_0 · · ·Tr_k;
```

Fig. 3. On-the-fly Model Checking, Including Trace Generation

3 Tuning Model Checking to Design Exploration

One of the challenges in the concept of exploration is to provide the user with as much information as possible, so that it is quick and easy to access. In this section we describe how to tune model checking so as to meet those challenges. The algorithmic features we describe include:

- The ability to provide a *maximal partial trace* (3.1), in the case a full trace for the specified path does not exist.
- An algorithm to produce *disjoint multiple traces* (3.2), different from each other for each path. This gives the user more insight about the design.

– An *interactive mode* (3.3) of the model checker, providing the user with the ability to obtain more immediate information about the model.
– The integration with *simulation* (3.4), which, when used for exploration purposes, is important for the success of the concept.

3.1 Maximal Partial Trace

When using model-checking techniques as a means for design exploration, the user always expects to get a trace as a result of the search. Traditional model checking algorithms will only produce a trace when such exists for the entire path that was specified. In case only part of the specified path exists in the model, no trace will be provided for the user. We show how to produce a maximal *partial trace* (in terms of *events* encountered), when no full trace exists for the given path.

For this purpose, we introduce auxiliary formulas, called *event formulas*, to help determine which events have been encountered at each iteration of the reachability search. An event formula is generated for each event, apart from the final one. We use the special structure of the path specification and the automaton built for it to derive the event formulas.

As described in [5], the automaton is built in such a way, that a move from one state to another is conditioned by a Boolean condition C. For every state s with condition C, which does not have a self loop, we produce a formula $EF(s \wedge C)$.

For example, consider again the path specification given in Figure 1 and its automaton shown in Figure 2. In addition to the formula $EF((aut = 6) \wedge (ma = 1))$ which specifies the full path, we generate the following formulas:

1. $EF((aut = 2) \wedge (ma = 4))$ and
2. $EF((aut = 4) \wedge (ma = 6))$

These formulas state that *event 1* has been reached and *event 2* has been reached respectively.

The auxiliary formulas are checked on-the-fly, while searching the reachable state space. Thus, when the model-checker determines that a full trace does not exist for the specified path, it produces the maximal *partial trace* available in the model. The enhanced algorithm is given in Figure 4 below. The terms ef_1..ef_n represent the BDDs [7] of the event formulas.

Progress Indication. Even when design exploration is applied to relatively small blocks, the search for a trace may take a long time, during which the user has very little information about the progress of the search. In model checkers such as SMV [15] and RuleBase [4], progress information is given in terms of *iteration*, which tells very little to the non-expert user. Information in terms of *events* specified would be of better value to such users.

Lines 9–13 in the Partial Trace algorithm described in Figure 4, show how *event formulas* are checked on-the-fly. When an event is found, the user is notified about it, as shown in line 12.

Taking this one step further, the user is granted the opportunity to interrupt the search, and be presented with the maximal partial trace currently available. Note that a partial trace produced during the search, is not necessarily a prefix of the full trace produced for the given path specification. In fact, it may be the case that the partial trace can not be extended to a full trace.

```
1 reachable = new = initialStates;
2 i = 0; maxe = 0
3 while ((new ≠ ∅)&&(new ∩ found = ∅)) {
4       S_i = new;
5       i = i+1;
6       next = nextStateImage(new);
7       new = next \ reachable;
8       reachable = reachable ∪ next;
9       for (j = n downto maxe+1) do {
10          if (new ∩ ef_j ≠ ∅) {
11              maxe = j; doughnut = i;
12              print “Event "maxe" encountered on cycle "doughnut"“
13              break; (from ‘for‘ loop)
14          }
15      }
16 if (new = ∅ && maxe = 0) {
17      print “No trace exists for this path“;
18      return;
19 }
20 else if (new = ∅) {  maxe > 0
21      print “No full trace exists. Producing trace until event "maxe"”;
22      found = ef_maxe; k = doughnut;
23 }
24 else {
25      k = i-1;
26      print “Trace found on cycle k”;
27 }
28 good = S_k ∩ found;
29 while (k>=0) {
30      Tr_k = choose one state from good;
31      if (k>0) good=pred(Tr_k)∩S_k-1;
32      k = k-1;
33 }
34 print “Trace is:” Tr_0 · · ·Tr_k;
```

Fig. 4. On-the-fly Model Checking, with Partial Trace Generation

3.2 Disjoint Multiple Traces

In the exploration paradigm, the user would benefit from being presented with many possible instances of the specified path. Moreover, the produced traces should be made as different from one another as possible, in order to provide the user with as much information as possible. In traditional model checking, a single trace is produced for each specification. In this section we demonstrate how to produce many traces (the number of traces can be specified by the user), while maintaining many variations. Lines 16–22 in Figure 3, give the single trace production algorithm.

The Disjoint Multiple Traces algorithm presented here is hueristic, and therefore is not guaranteed to find disjoint traces, However, our practical experience shows that it almost always does.

For this algorithm, we define the *distance* between a state s and a set of states P, to be the average of the Hamming distances between s and each of the states in P. Given two sets of states Q and P, our algorithm looks for a state $s \in Q$ which is as *far* as possible (has the biggest distance) from P.

Let $S_0, ..., S_k$ be the list of "new" sets, as appear in Figure 3. For each S_i, we keep a BDD P, of all states from S_i, already given in a produced trace. Given a BDD of a set of states Q (A subset of S_i) from which a state should be chosen, we replace line 18 in Figure 3 by choosing a state from Q as *far* as possible form P, if such a state exists. The heuristic algorithm for finding a *far* state is given in Figure 5.

Briefly, the algorithm works as follows: if $level(Q) < level(P)$, we recursively find a *far* state in the left hand side of Q, and in the right hand side of Q. We then compare their distance from P, and return the farther state, extended with the appropriate value of the current level. If $level(P) < level(Q)$, we recursively find a state in Q which is *far* from the left side of P and a state which is *far* from right side. We return as above. If $level(Q) = level(P)$, we compute $PP = P \rightarrow left \vee P \rightarrow right$ and recursively find a state in $Q \rightarrow left$ and in $Q \rightarrow right$ which are *far* from PP. If both sides return zero (no different state was found), we recursively find a state in $Q \rightarrow left$ which is *far* from $P \rightarrow left$, and a state in $Q \rightarrow right$ which is *far* from $Q \rightarrow right$. We return as above. We then add the chosen state to P by a disjunction of the BDDs.

The complexity of the algorithm is $O(N^2 * 2^m)$, where N is the size of the BDDs, and m is the number of BDD levels.

3.3 Interactive Design Exploration

The main purpose of this feature is to let the user gain additional information about the design block as quickly as possible. In *interactive mode* the model checker does not terminate after finding the desired traces. It saves all information in memory (list of doughnuts, reachable state set, provided traces etc.), and interactively serves new requests coming from the user, thereby providing the user with new information as desired. The primary types of user requests supported by our experimental exploration system are presented below.

Additional Cycles. This type of request specifies the number, N, of additional cycles required by the user as an extension of the current trace. The algorithm then performs N forward steps from the final state of each previously produced trace. This is possible as the traces and transition relation are stored in the model checker. When performing forward steps, we apply the same algorithm as in section 3.2 to choose the new states different from the previous, and thus more interesting to the user.

Additional Traces. This type of request allows the user to ask for N more traces, different from all the others already produced. Since all previously produced traces for the current path are saved inside the model checker, we simply apply the algorithm given in 3.2 N times to produce the desired traces.

Longer Trace. This type of request allows the user to ask the model checker to search for a longer trace than those already produced. Recall that in an on-the-fly algorithm, we are given an $EF(p)$ type formula, and we search for the first state in which p holds. Thus, the trace produced is the shortest available. In order to provide a longer one, we delete the set of *good* states from the doughnut in which it was first found, and continue reachability analysis, searching for the next time p holds.

```
1 function find_diff_state(P, Q) {
2   if ( level(Q) < level(P) ) {
3     (state0, dist0) = find_diff_state(P, Q → left);
4     (state1, dist1) = find_diff_state(P, Q → right);
5     if ( dist0 > dist1 )
6       return ( new_bdd(level(Q), state0, ZERO), dist0 );
7     else
8       return ( new_bdd(level(Q), ZERO state1), dist1 );
9     }
10  }
11  if ( level(Q) > level(P) ) {
12    (state0, dist0) = find_diff_state(P → left, Q);
13    if dist0 > 0 dist0 ++;
14    (state1, dist1) = find_diff_state(P → right, Q);
15    if dist1 > 0 dist1 ++;
16    if ( dist0 > dist1 ) {
17      return ( new_bdd(level(P), state0, ZERO), dist0 );
18    else
19      return ( new_bdd(level(P), ZERO, state1), dist1 );
20    }
21  }
22  if ( level(Q) = level(P) ) {
23    PP = or_bdd(P → left, P → right);
24    (state0, dist0) = find_diff_state(PP, Q → left);
25    (state1, dist1) = find_diff_state(PP, Q → right);
26    if (dist0 = 0 and dist1 = 0) {
27      (state0, dist0) = find_diff_state(P → left, Q → left);
28      (state1, dist1) = find_diff_state(P → right, Q → right);
29    }
30    if ( dist0 > dist1 ) {
31      return ( new_bdd(level(P), state0, ZERO), dist0 );
32    else
33      return ( new_bdd(level(P), ZERO, state1), dist1 );
34    }
35  }
36 }
```

Fig. 5. Choosing a state in BDD Q which is *far* from all states in BDD P

3.4 Reconstruction and Simulation

Contemporary model checking tools apply *reduction* algorithms [4] before model checking search starts. This is done to reduce the size of model to be actually model-checked, and is an efficient and important step for industrial model checkers. In many cases, this phase may reduce a large portion of the design, leaving the model checker with a much smaller task. The reduction algorithms usually applied are *safe* ones. That is, the reduced variables are indeed redundant, and are not needed for the evaluation of the specification. However, once a trace is produced, the reduced variables are often needed for the analysis of the trace. Specifically this is true when the tool is used by non-experts, as done in the exploration paradigm.

In order to solve this problem, we need to *reconstruct* the behavior of the reduced variables. To do so, we integrate a *simulator* with the model checker. When a trace is produced, it is first sent together with the original design and environment to the simulator. The simulator uses the values of the signals in the trace to calculate the values of all other signals. Thus, when the trace is presented to the user, all signals values are available and can be viewed.

A very useful feature of the simulation engine underlying our experimental system is that it allows direct manipulation of signal values: once a trace is produced and displayed, the user can edit input variables in the trace, and explore the different scenarios made possible by the introduction of these changes.

4 Experience

The methodology and algorithms described in this paper have been implemented in *PathFinder*, a tool which has been used by IBM designers in the past year. Initial experiments reveal designer acceptance at a significantly higher level than that observed with traditional application of model checking, and even higher than that observed with simulation tools for block-level testing. Typically, PathFinder is used on a freshly written logic of a few hundred state variables for 3–4 days, and finds 10–15 bugs.

While PathFinder hides most of the difficult parts of model checking, the user still must learn new concepts which are different from those of simulation. The ability to specify events over output and internal signals, and allow the tool to automatically find the right *input* behavior, is a new concept that people need to become accustomed to. The concept of non-determinism is also confusing at first. Experience shows though, that two hours of education and one day of supervision are sufficient for a designer to be able to efficiently use the tool.

5 Conclusion and Future Directions

In this paper we presented design exploration through model checking techniques. We have given several algorithms and features to support the new concept, searching for ways to improve the facilities provided by the mode checking tool, rather than attacking the inherent size problem of these methods. We believe that the exploration flavor of model checking techniques will open up new directions, making model checking accessible to a larger community, and bringing new interest in model checking techniques.

We believe the *interactive mode* described in section 3.3 to be a promising method. In order to provide the user with further insight, other options can be developed. For example, allowing the user control over the chosen traces, not just making them as different as possible, but letting the user direct the choice. Another is improving the ability to provide a longer trace, beyond the current reachable-states search iterations.

In the future, we plan to address the problem of *regression*. Currently the user must analyze every trace manually to determine if the behavior demonstrated is legal. A way to store and verify expected results in future runs could help extend the use of the exploration tool.

Acknowledgments

We thank Yael Abarbanel-Vinov, Roy Armony, Eli Berger, Eli Dichterman and Leonid Gluhovski, for their contribution to the implementation of algorithms described in this paper. We thank Sabina Joseph for her help in the deployment of PathFinder.

References

1. N. Amla, E.A. Emerson, R.P. Kurshan, K.S. Namjoshi. Model Checking Synchronous Timing Diagrams. In *Proc.* 3^{rd} *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, LNCS 1954, pages 283–298, 2000.
2. J. Baumgartner, T. Heyman, V. Singhal, and A. Aziz. Model checking the IBM Gigahertz Processor. In *Proc.* 11^{th} *International Conference on Computer Aided Verification (CAV)*, LNCS 1633, pages 72–83. Springer-Verlag, 1999.
3. J. Baumgartner, A. Tripp, A. Aziz, V. Singhal and F. Andersen. An Abstraction Algorithm for the Verification of Generalized C-slow Designs. In *Proc. of* 12^{th} *International Conference (CAV)*, 2000, pp. 5-19.
4. I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: an industry-oriented formal verification tool. In *Proc.* 33^{rd} *Design Automation Conference (DAC)*, pages 655–660. Association for Computing Machinery, Inc., June 1996.
5. I. Beer, S. Ben-David, A. Landver, On-The-Fly Model Checking of RCTL Formulas. In *Proc. of* 10^{th} *International Conference (CAV)*, 1998, pp. 184-194.
6. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, Y. Rodeh. The Temporal Logic Sugar. *submitted to CAV* 2001.
7. R.E. Bryant, Graph-based algorithms for boolean function manipulation, In *IEEE Transactions on Computers, C-35(8)*, 1986.
8. E.M. Clark and E.A. Emerson. Characterizing Properties of Parallel Programs as Fixed-point. In *Seventh International Colloquium on Automata, Languages, and Programming*, Volume 85 of LNCS, 1981.
9. G.F. De Palma, A.B. Glaser, R.P. Kurshan, G.R. Wesley, Apparatus for defining Properties in Finite-State Machines. *US Patent 6,966,516*, October 1999.
10. Á. Eiríksson. The formal design of 1M-gate ASICs. In *Second International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, LNCS 1522, pages 49–63. Springer-Verlag, 1998.
11. C. Eisner, R. Hoover, W. Nation, K. Nelson, I. Shitsevalov, and K. Valk. A methodology for formal design of hardware control with application to cache coherence protocols. In *Proc.* 37^{th} *Design Automation Conference (DAC)*, pages 724–729. Association for Computing Machinery, Inc., June 2000.
12. K. Fisler Timing Diagrams: Formalization and Formal Verification, In *Journal of Logic, Language and Information 8(3)*, 1999.
13. A. Goel and W. Lee. Formal verification of an IBM Coreconnect Processor Local Bus arbiter core. In *Proc.* 37^{th} *Design Automation Conference (DAC)*, pages 196–200. Association for Computing Machinery, Inc., June 2000.
14. D. Long. Model Checking, Abstraction and Compositional Verification. *Ph.D. Thesis*, CMU, 1993.
15. K.L. McMillan. Symbolic Model Checking. *Kluwer Academic Publishers*, 1993.
16. K.L. McMillan. A Methodology for Hardware Verification using Compositional Model-Checking. In *Science of Computer Programming*, 37(1-3):278-309 (2000)
17. K. Ravi, F. Somenzi. Hints to Accelerate Symbolic Traversal. In *Proc of CHARME*, 1999, pp. 250-264.
18. B.D. Winters, A.J. Hu. Source-Level Transformations for Improved Formal Verification. In *IEEE International Conference on Computer Design*, 2000.

This article was processed using the LaTeX macro package with LLNCS style