CONSTRUCTING CHECKERS FROM PSL PROPERTIES

Stefan Valentin Gheorghita[†‡] and Radu Grigore[‡]

†*Eindhoven University of Technology, PO Box 513,*
5600MB, Eindhoven, The Netherlands
‡NoBug Consulting, Bucharest, Romania
Email: s.v.gheorghita@tue.nl, radu.grigore@nobugconsulting.ro

Abstract: Model checking and simulation are the main techniques widely used in hardware verification. The past years trend is to bring together these two verification techniques in order to employ knowledge and tools produced by one to help the other. This paper describes a tool that translates properties written in PSL, a model checking language, into checkers written in languages suitable for simulation. The tool has two main focuses: first, retargetability, and second, simplicity, efficiency and clarity of the resulted checkers. However, its speed is in the same order of magnitude with those of commercial tools already existing on the market.

Keywords: Hardware Verification, Model Checking, Simulation, Hardware Design

1 INTRODUCTION

Two main techniques are currently widely used in the verification of hardware design: model checking and simulation. The model checking technique is applied mainly to small and medium size designs. The size limitation comes from the complexity of the algorithms used for verification. As a result, these languages are not designed for writing large test environments. They do inherit code structure constructs from HDLs (Hardware Description Languages), but these are inferior to those found in general purpose programming languages like Java. Instead, they are very expressive, in the sense that the code for expressing a seemingly complicated property is very short, compared to languages used in simulation-based techniques. The algorithms used for simulation checking are much more efficient because, by definition, this technique is not intended to be exhaustive. The languages offer much smaller "bricks" and advanced ways in which they can be "glued" to form a big edifice, much like general purpose programming languages.

A trend (Abarbanel and et al 2000) has emerged in the hardware verification community to work towards the convergence of these two verification technique categories. The idea is to employ knowledge and tools produced by one domain to help the other.

A hardware testing engineer receives a HDL design and a specification, written in English, that describes what the design does. His job is to verify the compatibility between the design ("how") and the specification ("what"). To do this, he creates a simulation environment written in a verification language. The purpose of the simulation environment is to provide inputs, to *check* the outputs and, sometimes, internal signals of the design under test. The parts that check the behavior are named "checkers" (Geist and et al 1999).

Therefore, these checkers are created from an English description. Instead of using a general purpose verification language to describe them, it is better to use a formal language that resembles the informal language used in the specification document. PSL (Property Specification Language) tries to do that. Its syntax and semantics are described in (Accellera 2003).

This paper describes a tool that translates properties written in PSL into checkers written in languages suitable for simulation-based environments. The tool can generate four kinds of automata (deterministic or not, with counters or not). This allows taking advantage of special built-in capabilities of describing automata that output languages might have. For example, if an output language supports direct description of nondeterministic automata, then constructing such an automaton saves space. If the output language allows only deterministic behavior (e.g. Verilog), then a deterministic automaton can be built. Currently, two different languages are supported, e (Verisity 2002) and Verilog (IEEE 2001).

The main characteristics of this translator are: (i) it is

easily retargetable to other (simulation) languages, (ii) it is fast, therefore, can be used in a commercial setting, (iii) its generated checkers are small, efficient and easily understandable by humans

Section 2 presents similar existing tools. Section 3 and Section 4 show the process by which the algorithms are derived, by giving a few examples. The automata constructed by these algorithms are optimized (Section 5). The architecture (Section 6) and the performance (Section 7) are presented towards the end of the paper.

2 RELATED WORK

The first tool designed to generate checkers from PSL properties was FoCs (Abarbanel and et al 2000), produced by IBM Research Laboratory in Haifa. It generates checkers written in Verilog, VHDL and C++. An advantage of FoCs over our tool is that it can process the GDL and Verilog modeling layers. The output generated by FoCs is more compact than ours because it takes full advantage of Verilog's [1] expressive power. Our tool is easily retargetable because it requires only very basic functionality from the target language. An interesting side-effect is that, although the output might be bigger, it is much more easily comprehensible by humans.

Another study was done at Cambridge (Gordon *,et al.*, 2003; Gordon 2004) on generating checkers using automated reasoning in HOL. This tool is too slow to be used in practice, but it can be used to test or prove correctness of tools like FoCs and the one described in this paper, which are not accompanied with fully formal proofs of the algorithms used.

3 PSL AND AUTOMATA SEMANTICS

3.1 Introduction to PSL Semantics

The standard (Accellera 2003) defines the semantics of PSL properties[2] in terms of traces. In general, these can be tree-like and unbounded. Since we restrict our discussion to properties that can be verified by simulation, we consider only linear finite length traces.

A *trace* t is a finite sequence of states; formally $t = (s_0, \ldots, s_{n-1})$ where $s_i \in S$, the set of all possible states of a circuit. A state determines the value of *all* the signals in the circuit for some clock. The design is considered to be digital and synchronous: signals have only two possible values (*asserted/true* and *not asserted/false*) and the time evolution is discrete.

For example, let us consider an inverter with input i and output o. For this circuit there are four possible states s_{io}. A valid trace is:

$$t_0 = (s_{01}, s_{10}, s_{10}, s_{01}, s_{10}) \tag{1}$$

An invalid trace is:

$$t_1 = (s_{00}, s_{10}, s_{10}, s_{01}, s_{10}) \tag{2}$$

A PSL property that describes an inverter, specifying the family of "correct" traces that contains t_0 but not t_1, is[3]:

$$always\, i \neq o; \tag{3}$$

The semantics of PSL properties is given by defining the $\models$ relation. For a trace t and a PSL property p the expression $t \models p$ evaluates to true if and only if t belongs to the family defined by p. For example, if we denote by p the property defined in equation 3 then we have:

$$(t_0 \models p) = \text{TRUE} \tag{4}$$

$$(t_1 \models p) = \text{FALSE} \tag{5}$$

According to the PSL Language Reference Manual (LRM), every construction that is syntactically correct has well defined semantics. The meaning of the basic operators is given by defining $\models$. The other operators can be reduced to basic operators by applying a set of rewritting rules.

Before giving a few definitions from the standard we need to introduce a few notations for operations on traces. Concatenation is denoted by writing the operands next to each other: $t_1 t_2$. For simplicity we use the same notation for a state and a trace with one state. We can also "cut" a part of a trace: if $t = (s_0, \ldots s_{n-1})$ then $t^{i,j} = (s_i, \ldots, s_j)$. As a shorthand, we write t^i instead of $t^{i,n-1}$. The length of a trace is denoted $|t| = n$. Some example definitions of basic operators are the following:

- $t \models b$ means that $|t| \geq 1$ and $b \in s_0$, where b is a boolean signal or a boolean expression constructed from signals; if b appears inside a SERE then there is the additional requirement $|t| = 1$
- $t \models p_1; p_2$ means that there exists t_1 and t_2 such that $t = t_1 t_2$, $t_1 \models p_1$ and $t_2 \models p_2$
- $t \models p_1 : p_2$ means that there exists t_1, t_2 and s such that $t = t_1 s t_2$, $t_1 s \models p_1$ and $s t_2 \models p_2$
- $t \models \text{X!}\, p$ means that $|t| > 1$ and $t^1 \models p$
- $t \models [p_1 \text{ U } p_2]$ means that there exists $0 \leq k < |t|$ such that $t^k \models p_2$ and for all $0 \leq j < k$ we have $t^j \models p_1$
- $t \models p_1 \mapsto p_2$ means that for all $0 \leq j < |t|$ $t^{0,j} \models p_1$ implies that there exists $j \leq k < |t|$ such that $t^{j,k} \models p_2$ or for all $j \leq k < |t|$ there exists a trace t' such that $t^{j,k} t' \models p_2$

Some operations defined by reductions (see (Beer and et al 2001) and (Accellera 2003) for a complete list) are:

- $p[*i] \equiv p; p; \ldots; p$ (i times)
- $always\, p \equiv \text{G}\, p \equiv \neg\, \text{F}\, \neg p$
- $\text{F}\, p \equiv [\text{TRUE U } p]$
- $next\, p \equiv \text{X}\, p \equiv \neg\, \text{X!}\, \neg p$

Using these rules we can find the formal semantics of the property given in 3. First, the reduction rules are applied. The result is:

$$\neg[\text{TRUE U } (i = o)] \tag{6}$$

[1] Verilog is the common output language between our tool and FoCs.

[2] The word "property" is used in a sense that includes SEREs (Sugar Extended Regular Expressions). The distinction between these two terms is not essential for the purpose of this paper.

[3] The symbol $\neq$ between booleans is used to denote the "exclusive or" operation. The symbol $=$ will be used to denote "not exclusive or".

Afterward, the definitions of basic operators are applied. After some logic manipulation $t \models always(i = o)$ becomes:

$$[\forall k.0 \leq k < |t| : (i \neq o) \in s_k] \quad (7)$$

3.2 Automata Semantics

The semantics on traces of deterministic finite automata (DFAs) with accepting states is standardized. The DFA guarantees that there is exactly one active state at each moment of time. A DFA has only one initial state and for each of its states and each possible circuit state there is exactly one active transition. Under these conditions, we say that automaton A is similar with property p if for all traces t the automaton reaches an accepting state if and only if $t \models p$. We write $A \sim p$. The complement automaton, defined by $B \sim \neg p$, is obtained from A flipping the accepting flag on each state.

Given such an automaton, we can verify if a property holds for a circuit by simulating the circuit, driving the automaton with the circuits signals and looking at the end of the simulation at the state the automaton is in. If it is an accepting state, then the property holds. Otherwise, it does not. However, this can hardly work in practice. For most of the properties it is possible to say before the end of the simulation if a property holds or not. Consider the example in equation 3. If at some clock k we find that q does not equal the value of d at the previous clock, there is no need to wait for the end of the simulation: we can say right away that the property does not hold. A state is *unconditionally* accepting only if it is accepting and all reachable states are accepting. A state is unconditionally non-accepting only if it is not accepting and neither is any reachable state.

In order to verify if a property holds for a simulation, we can construct the complement automaton and drive it by the circuit signals. If at some point in time we enter an unconditionally accepting state, we can signal an error immediately, without waiting for the end of the simulation. This is much more informative for the user who can see exactly when the circuit did not behave as expected. If at some point in time we enter an unconditionally non-accepting state, then we can stop looking: the property certainly holds for all possible traces with the current prefix. Otherwise, we need to check at the end of the simulation if the automaton is in an accepting state or not: if it is, then we signal an error.

In the following, we extend the automata model (1) by allowing non-determinism and (2) by introducing counters in order to increase the performance, while keeping the semantics unchanged.

A non-deterministic automaton is one that guarantees that, at any time, there is *at least* one active state. Such an automaton is said to be similar to a property p if for all traces t it has at least one accepting state if and only if $t \models p$. There is no easy way to find the complement of such an automaton: the equivalent deterministic automaton (by the usual conversion from non-deterministic to deterministic automata) must be obtained and then its complement is computed. The conversion from NFA to DFA should be avoided as much as possible since, in the worst case, the size of the resulted DFA is exponential in the size of the NFA (Hopcroft and Ullman 1979).

The same definitions for unconditionally accepting and non-accepting states apply on NFAs. However, there is a subtle interpretation difference. Suppose we are given the NFA similar to $\neg p$ and we want to check the validity of property p for some simulation. If at some point in time the automaton has one active unconditionally accepting state, then we can immediately say that the property does not hold. Otherwise, if at some point in time *all* the active states are unconditionally non-accepting, then we can be certain that the property holds, no matter what else happens until the end of the simulation. Otherwise, we need to look at the active states at the end of the simulation: if at least one of them is accepting, then the property does not hold for the simulation.

The addition of counters drastically decreases the size of the automata generated for properties like $q[*65000]$. The idea is to identify an automaton state with a node in a graph *and* a set of counters, each having a (natural number) value. In theory, this makes the automaton infinite. The representation of an automaton without counters, similar to the mentioned property, needs at least $1/2$ MB of memory[4], while one with counters requires only a few bytes. The transition conditions can now contain boolean terms in the form of $c == 0$, where c is the name of a counter. They can also contain commands that modify the value of a counter, either by setting it to a constant ($c = \text{CT}$) or by decrementing it ($--c$). The accepting / non-accepting status of a state is determined by its corresponding graph node, i.e. the counters values do not discriminate between accepting and non-accepting states.

An automaton with counters can always be transformed to another one without counters by making explicit the state families corresponding to graph nodes. Each state family has $n_1 n_2 \cdots n_C$ members where n_i is the maximum value of counter i plus one, and C is the number of counters used.

The automata used inside our tool are generally NFAs with counters. When the complement of an automaton is needed, first its counters are removed and then it is converted to a DFA. The complement of the a DFA is easy to compute. Constructing the automaton as a DFA in the first place is not a viable solution, since the constraints on the combining algorithms would make them too complicated.

4 IMPLEMENTATION OF AUTOMATA COMBINING OPERATORS

The first subsection describes the building blocks from which automata are constructed and the other subsections

[4]There are 65000 states, and for each state at least 50 bits are used (16 bits for the state name, 32 bits for the name of the 2 transactions' destinations and 2 bits for transactions' conditions).

present the algorithms corresponding to few of the PSL operators supported by our tool (see Appendix B of (Gheorghita 2003)). As an exhaustive presentation of all of them would exceed the limits of this paper, we consider the following criteria for choosing some: to cover many automaton model features, to be non-standard operators (not part of regular expressions), to have interesting interdependencies between them and to illustrate a variety of algorithm design techniques that we have used.

4.1 Primitive Automata

The simplest property is a boolean expression: constant ($t \models$ TRUE and its complement$t \models$ FALSE) or non-constant ($t \models b$). Since these properties automata are deterministic, it is easy to see their complements.

4.2 Fusion

One of the extensions to regular expressions is the fusion operator. It is syntactically denoted using a colon (":"). The semantics is given by:

$$t \models p_1 : p_2 \equiv [\exists k.0 \leq k < n : t^{0,k} \models p_1 \wedge t^k \models p_2] \quad (8)$$

In order to find the rules for constructing the similar automaton, we consider two paths: one from the automaton similar to p_1 and one from the automaton similar to p_2.

$$q_0^1 \xrightarrow{s_0} q_1^1 \xrightarrow{s_1} q_2^1 \cdots q_k^1 \xrightarrow{s_k} q_{k+1}^1; \\ (q_0^2 \xrightarrow{s_k} q_1^2 \xrightarrow{s_{k+1}} q_2^2 \cdots q_{n-k-1}^2 \xrightarrow{s_{n-1}} q_{n-k}^2) \quad (9)$$

Here, q_i^1 are the states of the automaton similar to p_1 and q_i^2 are the states of the automaton similar to p_2. Both paths start from an initial state and end in an accepting state. From definition 8, if two such paths exist, then there must be a path in the automaton similar to $p_1 : p_2$ that starts from an initial state, ends in an accepting state and is driven by t.

We can do this by connecting state q_k^1 to state q_1^2 with a transition that is active only when both $q_k^1 \rightarrow q_{k+1}^1$ and $q_0^2 \rightarrow q_1^2$ are active. The role of state q_k^1 can be taken by any state with a transition to an accepting state in the first automaton. The role of the q_1^2 state can be taken by any state with a transition from an initial state in the second automaton. In the end, we obtain the following algorithm:

FUSION(A, B)

```
1  for each t_A ∈ transitions[A]
   such that type[target[t_A]] = ACCEPT
2      do for each t_B ∈ transitions[B]
          such that initial[src[t_B]] = TRUE
3             do add new transition t
4                src[t] ← src[t_A]
5                target[t] ← target[t_B]
6                cond[t] ← cond[t_A] ∧ cond[t_B]
7  for each n ∈ nodes[A]
8      do if type[n] = ACCEPT
9         then type[n] ← NOT-ACCEPT
10 for each n ∈ nodes[B]
11     do initial[n] ← FALSE
```

This algorithm has three important steps: add transitions from A to B (lines 3–6), make the accepting states of A non-accepting (the **for** at line 7) and make the initial states of B normal states (the **for** at line 10). The lines 3–6 are executed for each pair (t_A, t_B) such that t_A is a transition of automaton A that ends in an accepting state and t_B is a transition of automaton B that starts from an initial state. A new transition t with the same source as t_A and the same destination as t_B is added. The condition on t is the disjunction of conditions of t_A and t_B.

4.3 Fixed Times Consecutive Repetition

Fixed times consecutive repetition operator, introduced in section 3.1, is denoted by $[*i]$, where i represents the number of repetitions. Instead of using the presented reduction rules, which will increase dramatically the size of the resulted automaton, our algorithm uses the counter's feature of the automaton:

COUNTER-FIXED-REPETITION(A, i)

```
1  if i = 0
2     then return a new automaton with
            only one accepting initial state
3  create counter c
4  set initial value of c to i
5  for each t ∈ transitions[A]
   such that type[target[t]] = accept
6        do for each n ∈ nodes[A]
            such that initial[n] = TRUE
7              do add new transition t_new
8                 src[t_new] ← src[t]
9                 target[t_new] ← n
10                cond[t_new] ← cond[t] ∧ (c ≠ 0)
11                act[t_new] ← act[t], − − c
12          cond[t] ← cond[t] ∧ (c = 0)
```

The line 12 ensures that the resulting automaton will accept a trace only if i consecutive segments of the trace have been accepted.

4.4 Weak Suffix Implication

Until now, we have tried to find only an automaton similar to a property p. We need also to find the automaton similar to the negated property as well.

The algorithm for weak suffix implication starts from $A \sim p$ and $B \sim q$ and obtains the automaton similar to $\neg(p \mapsto q)$. In order to derive the algorithm, we start from the formal definition:

$$t \models \neg(p \mapsto q) \equiv [\exists j.0 \leq j < |t| : t^{0,j} \models p \wedge \\ [\forall k.j \leq k < |t| : t^{j,k} \models \neg q] \wedge \\ [\exists k.j \leq k < |t| : [\forall t' : t^{j,k}t' \models \neg q]]] \quad (10)$$

Suppose we have already computed $C \sim \neg q$ from B. Cutting all outgoing transitions from non-accepting states in C gives an automaton that evaluates the first term of the conjunction. Marking as non-accepting all nodes that are *conditionally* accepting gives an automaton that evaluates

the second term of the conjunction. These two transformations must be done in order to create an automaton that evaluates the conjunction. The algorithm is:

WEAK-IMPLICATION(A, B)

```
1  C ← NEGATE(B)
2  for each t_C ∈ transitions[C]
3       do if type[src[t_C]] = NON-ACCEPT
4            then remove t_C
5  for each n_C ∈ nodes[C]
6       do if type[n_C] = NON-ACCEPT
7            then type[n_C]] = UNC-NON-ACCEPT
8          if type[n_C] = ACCEPT
9            then type[n_C]] = NON-ACCEPT
           ▷ Only UNC-ACCEPT remain accepting
10 FUSION(A, C)
```

The procedure NEGATE transforms an automaton into its complement by first transforming it from NFA to DFA and then flip the accepting state. Because there is no way to transform a NFA with counters into a DFA while preserving counters, these must not be present in automaton B. We can ensure this by providing alternative construction methods wherever counters are usually the best solution.

5 AUTOMATA OPTIMIZATION

The automata optimizations can be grouped into: transitions merging, reachability analysis, unconditional states melting and boolean conditions simplification.

Whenever there are two parallel transitions without commands attached to them (like setting or decrementing a counter), they can be fused together by using as the resulting condition the disjunction of the original conditions.

All states that are not reachable from at least an initial state can be removed together with their incoming and outgoing transactions.

All unconditional accepting states are equivalent; all unconditional non-accepting states are equivalent. This means that they can be simply melt together and all outgoing transitions replaced with a loop transition that is always active. Because unconditional tags are sometimes lost (see FUSION), they need to be restored by another reachability analysis.

In order to simplify the boolean conditions, Quine Mc Cluskey algorithm was used.

All these optimizations can be applied after each combination phase or only on the final result. Applying the optimizations after each combination step means a lot of extra work. Delaying until the result automaton is obtained has the potential of letting the automaton dimension to explode and, hence, the running time is even worse on complex properties. In order to improve speed performance, we are considering to develop smart heuristics that decide when to apply certain optimizations. These heuristics will replace the simple rules “always” and “just at the end”.

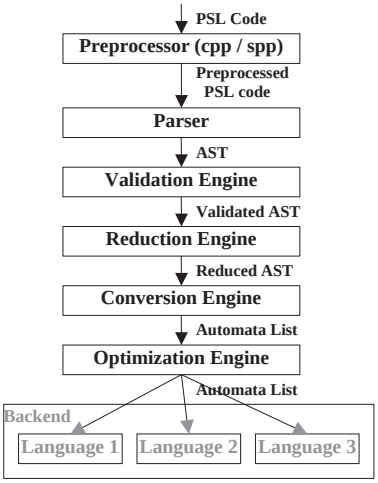

Figure 1. Tool architecture

6 TOOL ARCHITECTURE

Figure 1 shows the internal architecture of our tool. The tool consists of a sequence of several modules, each representing a translation phase. Different back-ends may be added to this tool in order to dump the automata into different target languages with specific syntax for automata support. PSL provides macro-processing capabilities to facilitate the definition of properties. It supports *cpp* style pre-processing directives (e.g. `#define`, `#ifdef`, `#else`, `#include` and `#undef`) and special macros for `%for` and `%if` that are used to conditionally or iteratively generate PSL properties.

Our tool uses *cpp* (FSF 2004) to handle the preprocessing directives. For the special macros, an internal preprocessor that extend *vpp* (Thaker 1996) is used. More information about its preprocessor may be found in Appendix C of (Gheorghita 2003).

The parser converts the input PSL files into the corresponding Abstract Syntax Tree (AST). During the conversion, some checks for the validity of input are done ((Gheorghita 2003) presents all these checks).

The **validation engine** executes all the checks which could not be handled during the parsing phase. These checks may be general checks or specific checks, which depend on the target language (on which the resulted automata are dumped). New checks are easy to be written and added to be executed by the engine.

The **reduction engine** applies the PSL reduction rules (presented in section 3.1) on the current AST. Its output is also an AST, in which only the basic PSL operators (and counters operators, if they are enabled) are used. The resulted AST may have more nodes than the original one, but the engine which converts it into automata is simpler because it must handle a smaller number of operator types.

The **conversion engine** generates automata based on the input AST. The resulted automata respect the semantics described in section 3.2. The engine traverses the AST in post-order. For every node reached, it builds an automaton, based on the automata generated for its children

Table 1. Tools comparison.

		Our tool	FoCs	HOL
Runtime	real	11.77s	3.10s	$>$ 1 day
	user	11.68s	0.60s	N/A
	system	0.03s	0.14s	N/A
States		316	310	N/A
Automata		172	77	N/A

and the information stored in the node. If the node is a tree leaf (has no children), a simple automaton is generated, as it is shown in section 4.1. The algorithm used to combine the children's automata is chosen based on the node internal information, which is an operator. Sections 4.2-4.4 present examples of operators and their associated algorithms.

The quality of conversion from PSL to automata is improved by the **optimization engine** that executes all the optimizations presented in section 5.

A backend is composed by several restrictions and a set of printing functions. The restrictions define what types of automata are supported by the backend language (e.g. Verilog supports only DFA). The tool adapts all the automata in order to respect the restrictions imposed by the backend. The printing functions dump the content of data structures into the backend language. Usually, having good knowledge about the destination language, a backend can be written in less than two days.

7 TOOL PERFORMANCE

In order to evaluate our tool, it was run to convert a commercial project that consists in 77 PSL properties. An AMD Athlon XP 2.2GHz, 1024MB RAM running Fedora Linux was used. Table 1 presents the comparison between our tool and the two existing tools, FoCs and a PSL embedding in HOL (Gordon *,et al.,* 2003).

The output generated by FoCs is very intricate and difficult to be understood by engineers, as it contains a complicated `if/then/else` structure where it is not easy to make the difference between automata's states and transitions. Opposite to this approach, our tool generates every automaton as a simple `switch` structure used to select the automaton state. The clarity of the generated code is enhanced by the fact that, instead of using a complicate automaton for a PSL property, multiple simple automata are used. This explains the big difference between the number of automata generated by our tool and those generated by FoCs, while that the number of states is almost the same. FoCs uses long and difficult to understand identifiers to name the generated structures, while we tried to select intuitive names for the identifiers.

The HOL PSL example is not very developed, as it accepts only a very simple subset of PSL grammar (actually, it supports only SEREs). Considering this, only 30% from our input could be processed by it. The execution time is very long (several days), compared with our tool's and FoCs'. Nevertheless, the generated output is very simple, efficient and easy to understand.

8 CONCLUSIONS AND FURTHER WORK

This paper presents a tool that translates PSL properties into checkers written in languages suitable for simulation-environments. Currently, two different languages are supported, e (Verisity 2002) and Verilog (IEEE 2001), but the tool is easily retargetable to other languages. Adding a backend for another language amounts merely to write a set of printing functions. Our tool runs five orders of magnitude quicker than HOL PSL scripts, and only four times slower than the commercial tool FoCs. Compared with FoCs, our tool's output is simpler and easier to be understood by engineers.

In the future, we plan to extend the PSL grammar subset supported by our tool and slowly migrate to the new semantics defined in the 1.1 version of the PSL standard. Furthermore, we are interested in improving the tool's execution time. One step in this direction is the development of the heuristic mechanism that decides when automata optimizations should be applied.

Another direction is to build up the level of confidence in our tool. We believe that testing the orrectness against a reference tool (e.g. HOL PSL scripts) has the potential of revealing bugs. Proving the correctness of the construction methods by automatic reasoning is another step to ensure the correctness of checkers generator. These two methods should complement each other.

REFERENCES

ABARBANEL, Y. and ET AL. (2000). FoCs: Automatic generation of simulation checkers from formal specifications. In *Proc. of the 12th Conf. Computer Aided Verification*. Springer LNCS, **vol. 1855**. 538–542.

ACCELLERA. (2003). Accellera property specification language reference manual version 1.01.

BEER, I. and ET AL. (2001). The Temporal Logic Sugar. In *Proc. of the 13th Conference in Computer Aided Verification*. Springer LNCS, **vol. 2102**. 538–542.

FSF. (2004). cpp, GNU C Preprocessor.

GEIST, D. and ET AL. (1999). A methodology for the verification of a system on chip. In *Proc. of the 36th Design Automation Conference*. 574–579.

GHEORGHITA, S. V. (2003). The art of translating Sugar to an Automata Language. M.Sc. thesis, CS Department, Politehnica Univ. of Bucharest, Romania.

GORDON, M. (2004). PSL semantics in higher order logic. In *5th Int Wsh. on Designing Correct Circuits*.

GORDON, M., HURD, J. and SLIND, K. (2003). Executing the formal semantics of the accellera property specification language by mechanised theorem proving. In *Proc. of the 12th Conf. on Correct Hardware Design and Verification Methods*. Springer LNCS, **vol. 2860**. 200–215.

HOPCROFT, J. and ULLMAN, J. (1979). *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publish Company.

IEEE. (2001). IEEE standard 1364-2001.

THAKER, H. M. (1996). vpp, a Verilog Preprocessor.

VERISITY. (2002). e language reference manual.