Executing the formal semantics of the Accellera Property Specification Language by mechanised theorem proving

Mike Gordon[1], Joe Hurd[1] and Konrad Slind[2]

[1] University of Cambridge Computer Laboratory, William Gates Building, JJ Thomson Avenue, Cambridge CB3 0FD, U.K.,
[2] University of Utah, School of Computing, 50 South Central Campus Drive, Salt Lake City, Utah, UT84112, USA

Abstract. The Accellera Property Specification Language (PSL) is designed for the formal specification of hardware. The Reference Manual contains a formal semantics, which we previously encoded in a machine readable version of higher order logic. In this paper we describe how to 'execute' the formal semantics using proof scripts coded in the HOL theorem prover's metalanguage ML. The goal is to see if it is feasible to implement useful tools that work directly from the official semantics by mechanised proof. Such tools will have a high assurance of conforming to the standard. We have implemented two experimental tools: an interpreter that evaluates whether a finite trace w, which may be generated by a simulator, satisfies a PSL formula f (i.e. $w \models f$), and a compiler that converts PSL formulas to checkers in an intermediate format suitable for translation to HDL for inclusion in simulation test-benches. Although our tools use logical deduction and are thus slower than hand-crafted implementations, they may be speedy enough for some applications. They can also provide a reference for more efficient implementations.

1 Introduction

We describe the implementation of two tools that work by applying theorem proving strategies to the formal semantics of the Accellera Property Specification Language (PSL [3]). The implementation method guarantees that the results are compliant with the standard. Accellera [2] is an industry consortium formed in 2000 by combining "Open Verilog International" and "VHDL International". PSL is being developed as a standard property language for both dynamic verification (e.g. simulation) and static verification (e.g. model checking) [8]. The design of PSL is based on IBM's Sugar language.

Previously we constructed a deep embedding of the Sugar semantics in higher order logic. Using the HOL theorem proving system we proved various general meta-theorems (see Section 2) and were able to provide some feedback and bug reports to the language designers [12, 11]. As the semantics evolved into the current standard we tracked the changes and made sure that our proofs in HOL still went through. Our semantics is believed to correspond faithfully to the official formal semantics in the PSL Manual, but we cannot be completely certain because the official semantics is expressed in a mixture of English and LaTeX.

Not only can theorem provers like HOL be used to prove meta-theorems, they can also be programmed to dynamically generate theorems for particular models and formulas. This provides a way of implementing tools that work deductively. The approach of having tools with 'HOL Proof Inside' has been explored by the Prosper project [7] and it is our goal to apply Prosper ideas to build verification tools that work with 'deduction from PSL semantics inside'. This paper describes some preliminary experiments.

PSL has four kinds of syntactic constructs: Boolean Expressions b, Sequential Extended Regular Expressions r (SEREs), Foundation Language (FL) formulas f and Optional Branching Extension (OBE) formulas.

The PSL Foundation Language (FL) contains standard future-time LTL formulas as well as less standard formulas that are composed out of regular expressions. Formula $\{r\}(f)$ is true if f holds at the last state of any sequence matching r; formula $\{r_1\} \mapsto \{r_2\}!$ is true if every sequence matching r_1 is followed by a sequence matching r_2. FL also has abort formulas f *abort* b that check f but aborts the checking if a state in which b is true is encountered, and clocking formulas $f@c$ that are true when f is true of the sequence of states consisting of only those states for which clock c holds.

The OBE is conventional branching time Computation Tree Logic (CTL). Hasan Amjad has built a symbolic model checker for OBE properties that uses BDD representation judgements applied to our semantics to calculate the truth-value of PSL properties with respect to Kripke structures. This is described elsewhere [4].

The semantics of SEREs specifies $w \models r$ to mean that a finite sequence of states w matches the regular expression r. Then semantics of FL formulas specifies $w \models f$ to mean that formula f holds of a path (i.e. a finite or infinite sequence of states). The detailed semantics is in Section 2. PSL also has a large number of operators that are defined in terms of the primitives. As we shall illustrate, they can be added by making definitions in HOL.

Using standard methods of semantic embedding, $w \models f$ can be viewed as a boolean term of higher order logic, and then automated proof by the HOL system can be applied. We have implemented a proof strategy to evaluate $w \models f$ where w is a specific finite path and f is a formula. Currently all formulas except aborts are covered (though a few special cases of $w \models f$ *abort* b can be evaluated). This strategy implements a tool that is useful for sanity checking that a property expresses what one expects: one can directly evaluate it on example paths and the result is guaranteed to correspond to the official semantics. Example paths can either be input directly as a sequence of states (a state is a set of atomic propositions), or can be captured from a simulation run (see Section 3.3 for examples). Evaluation is fast enough to be used on simple examples and provides a pedagogically useful animation of the semantics.

Our second tool, inspired by the IBM FoCs system [1], compiles a formula f (from a subset of PSL formulas) into a checker automaton that can be added to a simulation test-bench to detect when a property is violated. The checker is initially represented in an HDL-neutral format but can be 'pretty printed' into the syntax of particular HDLs. We have implemented a simple converter that generates Verilog. This provides a way of prototyping tools similar to FoCs, but which are guaranteed by construction to conform to the Accellara standard. Although generating a checker can be slow (seconds to minutes), the resulting HDL code can be efficient, and it is guaranteed to be equivalent to the PSL property it was compiled from. We think this compiler might be useful for debugging other property generators. Also, since the compilation is driven by symbolic execution, it can be tuned just by adding new theorems into the set of rules that are used.

The rest of this paper is as follows: Section 2 describes the Accellera property Specification Language (PSL) and its semantics in higher order logic; Section 3 presents our first tool, which evaluates $w \models f$ for a given w and f; Section 4 presents our second tool, a checker generator.

2 The Accellera Property Specification Language PSL

This section describes the semantics of the linear parts of PSL (boolean expressions, SEREs, FL formulas) and is a careful manual transcription of the official semantics in the Language Reference Manual [3] into the machine readable logic supported by the HOL system.

Boolean expressions are evaluated with respect to states. SEREs are evaluated with respect to finite sequences of states, and FL formulas with respect to finite or infinite sequences of states. A non-empty set P of atomic propositions is assumed given. A state is a subset of P, i.e. the set of propositions that are true in the state. If p ranges over P, then the syntax of boolean expressions b is:

$$\begin{array}{lll} b ::= & p & \text{(Atomic proposition)} \\ & \mid \neg b & \text{(Negation)} \\ & \mid b_1 \wedge b_2 & \text{(Conjunction)} \end{array}$$

This is represented in higher order logic by defining a new type (using a data type definition mechanism), parameterised on P, whose elements are boolean expressions. The semantics of boolean expressions are specified by defining $s \models b$, where $s \subseteq P$, by structural induction over the type of boolean expressions:

$$(s \models p \;=\; p \in s) \wedge (s \models \neg b \;=\; \neg(s \models b)) \wedge (s \models b_1 \wedge b_2 \;=\; s \models b_1 \wedge s \models b_2)$$

Here, and in what follows, the operator "$\models$" binds tightly, so that, for example, $s \models b_1 \wedge s \models b_2$ means $(s \models b_1) \wedge (s \models b_2)$ not $s \models (b_1 \wedge s \models b_2)$. The symbols $\neg$ and $\wedge$ are overloaded: the occurrence of $\neg$ in $\neg b$ is part of the boolean expression syntax of PSL, but the occurrence in $\neg(s \models b)$ is negation in higher order logic. Similarly $\wedge$ is overloaded: the occurrence in $b_1 \wedge b_2$ is part of the boolean expression syntax, but the other occurrences are conjunction in higher order logic.

2.1 Semantics of unclocked SEREs and FL formulas

In this section we do not specify the semantics of clocked SEREs $r@c$ and formulas $f@c$. These are described in Section 2.2.

The syntax of SEREs is represented in higher order logic by defining a new type whose elements represent SEREs. If r, r_1, r_2 etc. range over Sequential Extended Regular Expressions (SEREs) and b and c range over boolean expressions, then the syntax of SEREs is:

$$
\begin{array}{ll}
r ::= b & \text{(Boolean formula)} \\
\quad \mid \{r_1\} \mid \{r_2\} & \text{(Disjunction)} \\
\quad \mid r_1 ; r_2 & \text{(Concatenation)} \\
\quad \mid r_1 : r_2 & \text{(Fusion: overlapping concatenation)} \\
\quad \mid \{r_1\}\&\&\{r_2\} & \text{(Length matching conjunction)} \\
\quad \mid r[*] & \text{(Repeat)} \\
\quad \mid r@c & \text{(Clocking – semantics in Section 2.2)}
\end{array}
$$

The semantics of a SERE r is given by specifying $w \models r$ for every finite sequence of states w. This can be read as "word w is recognised by regular expression r".

Words are represented as lists. A list containing elements $e_0, \ldots, e_n$ is denoted by $[e_0; \ldots; e_n]$. Juxtaposition of words denotes concatenation (e.g. $w[s]w'$ is the concatenation of w, $[s]$ and w'). If $wlist$ is a list of lists then $\mathsf{Every}\ p\ wlist$ applies the predicate p to every element of $wlist$ and returns the conjunction of the result (e.g. in the semantics below $\mathsf{Every}\ (\lambda w.\ w \models r)\ wlist$ asserts $w \models r$ for every w in $wlist$) and $\mathsf{Concat}\ wlist$ denotes the concatenation of the lists in $wlist$ (e.g. $\mathsf{Concat}\ [[a;b];[c];[d;e;f]] = [a;b;c;d;e;f]$). The notation $|w|$ denotes the length of w (empty words have length 0) and w_i denotes the ith element of w counting from 0, so w_0 is the first element (note that subscripts on symbols not denoting lists are just subscripts). The input and output to HOL shown in this paper has been typeset using a HOL-to-Latex translator implemented by Keith Wansbrough. Applying this translator to the HOL semantics of SEREs yields:

$$
\begin{array}{l}
(w \models b \ = \ (|w| = 1) \wedge w_0 \models b) \wedge \\
(w \models r_1 ; r_2 \ = \ \exists w1\, w2.\ (w = w1\, w2) \wedge w1 \models r_1 \wedge w2 \models r_2) \wedge \\
(w \models r_1 : r_2 \ = \ \exists w1\, w2\, l.\ (w = w1\,[l]w2) \wedge w1\,[l] \models r_1 \wedge [l]w2 \models r_2) \wedge \\
(w \models \{r_1\} \mid \{r_2\} \ = \ w \models r_1 \vee w \models r_2) \wedge \\
(w \models \{r_1\}\&\&\{r_2\} \ = \ w \models r_1 \wedge w \models r_2) \wedge \\
(w \models r[*] \ = \ \exists wlist.\ (w = \mathsf{Concat}\ wlist) \wedge \mathsf{Every}(\lambda w.\ w \models r) wlist)
\end{array}
$$

It is hoped that this semantics requires no additional explanation. Interested readers can compare it to the semantics in the PSL Reference Manual [3, B.2.2.1].

The syntax of PSL Foundation Language Formulas (FL) is given below. The suffix "!" found on some constructs indicates that these are 'strong' (i.e. liveness-enforcing) operators. If the corresponding weak operator (which is written without the "!" suffix) can be defined in terms of FL formulas, then it is not included in the core and is regarded as an defined operator (e.g. $X f \ = \ \neg X! \neg f$ and $f@c \ = \ \neg(\neg f@c!)$). The distinction between strong and weak operators is discussed and motivated in the PSL Manual [3, Section 4.4.3].

The syntax is represented in higher order logic by defining a new type whose elements are formulas. The FL primitives listed below are redundant. For example, $\{r_1\} \mapsto \{r_2\}!$ and $X!\ f$ can be defined in terms of suffix implication.

$$
\begin{array}{lll}
f ::= & p & \text{(Atomic formula)} \\
& \mid \neg f & \text{(Negation)} \\
& \mid f_1 \wedge f_2 & \text{(Conjunction)} \\
& \mid X!\, f & \text{(Successor)} \\
& \mid [f_1\ U\ f_2] & \text{(Until)} \\
& \mid \{r\}(f) & \text{(Suffix implication)} \\
& \mid \{r_1\} \mapsto \{r_2\}! & \text{(Strong suffix implication)} \\
& \mid \{r_1\} \mapsto \{r_2\} & \text{(Weak suffix implication)} \\
& \mid f\ \mathit{abort}\ b & \text{(Abort)} \\
& \mid f@c! & \text{(Clocking – semantics in Section 2.2)}
\end{array}
$$

Paths can be either finite or infinite. The notation w^i denotes the i-th tail of w, i.e. the path obtained by chopping i elements off the front of w (so $w^0 = w$). The notation $w^{i,j}$ denotes the finite sequence of states from i to j in w, i.e. $w_i w_{i+1} \cdots w_j$. The juxtaposition $w^{i,j}w'$ denotes the path obtained by concatenating the finite sequence $w^{i,j}$ on to the front of the path w'. The HOL semantics of FL formulas is:

$$
\begin{array}{l}
(w \models b \ = \ |w| > 0 \wedge w_0 \models b) \wedge \\
(w \models \neg f \ = \ \neg(w \models f)) \wedge \\
(w \models f_1 \wedge f_2 \ = \ w \models f_1 \wedge w \models f_2) \wedge \\
(w \models X!\, f \ = \ |w| > 1 \wedge w^1 \models f) \wedge \\
(w \models [f_1\ U\ f_2] \ = \ \exists k \in (0 \mathinner{..} |w|).\ w^k \models f_2 \wedge \forall j \in (0 \mathinner{..} k).\ w^j \models f_1) \wedge \\
(w \models \{r\}(f) \ = \ \forall j \in (0 \mathinner{..} |w|).\ w^{0,j} \models r \Rightarrow w^j \models f) \wedge \\
(w \models \{r_1\} \mapsto \{r_2\}! \ = \ \forall j \in (0 \mathinner{..} |w|).\ w^{0,j} \models r_1 \Rightarrow \exists k \in (j \mathinner{..} |w|).\ w^{j,k} \models r_2) \wedge \\
(w \models \{r_1\} \mapsto \{r_2\} \ = \\
\quad \forall j \in (0 \mathinner{..} |w|). \\
\quad w^{0,j} \models r_1 \Rightarrow (\exists k \in (j \mathinner{..} |w|).\ w^{j,k} \models r) \vee (\forall k \in (j \mathinner{..} |w|).\ \exists w'.\ w^{j,k}w' \models r_2)) \wedge \\
(w \models f\ \mathit{abort}\ b \ = \ w \models f \vee w \models b \vee \exists j \in (1 \mathinner{..} |w|).\ \exists w'.\ w^j \models b \wedge w^{0,j-1}w' \models f)
\end{array}
$$

This semantics is a careful formalisation of the official semantics, with the exception that $|w| > 0$ has been added to the definition of $w \models b$. This addition ensures that formulas are defined for empty paths (the official semantics is undefined). The semantics for non-empty paths is unchanged.

2.2 Semantics of clocked SEREs and FL formulas

SEREs and formulas not containing "@" are called unclocked and the sets of unclocked SEREs and formulas the unclocked subsets. In the previous section only the semantics of the unclocked subsets were defined. This is called the unclocked semantics.

Clocked SEREs have the form $r@c$ and strongly clocked formulas the form $f@c!$, where c is a boolean expression that is true when the clock is asserted.

Weakly clocked formulas $f@c$ are defined by $f@c = \neg((\neg f)@c!)$. Intuitively, $w \models r@c$ and $w \models f@c!$ mean, respectively, that $w|_c \models r$ and $w|_c \models f$ where $w|_c$ is obtained from w by removing ('projecting out') all states in which c is false (i.e. restricting w to states in which c is true).

The formal semantics in the Reference Manual doesn't use projections, instead two separate semantics are given: the first one defines the semantics of all constructs (included clocked ones) directly, the second one provides a set of 'rewrites' that can be used to recursively eliminate all occurrences of "@", i.e. translate into the unclocked subsets.

The direct semantics is specified by recursively defining $w \models^{c} r$ and $w \models^{c} f$ for an arbitrary clock c, and then the semantics of a SERE r and formula f are $w \models^{\mathsf{T}} r$ and $w \models^{\mathsf{T}} f$, respectively, where T is the top-level clock which is always true. The top-level semantics with a clock c are $w \models^{\mathsf{T}} r@c$ and $w \models^{\mathsf{T}} f@c!$.

The rewrites semantics is formalised by first defining, for each clock c, a function $\mathcal{T}^c$ that maps an arbitrary SERE or formula into the unclocked subset. Thus $\lambda c.\ \mathcal{T}^c$ is a function mapping a clock c to a translation function $\mathcal{T}^c$ which has c as the clock context. The top-level clock is T, so the top-level translations of r and f are $\mathcal{T}^{\mathsf{T}}(r)$ and $\mathcal{T}^{\mathsf{T}}(f)$. The meanings of these can then be computed using the unclocked semantics in Section 2.1.

The definition of $\models^{c}$ is much more complex than the definition of $\models$, and we do not give it here. However, we have formalised it in higher order logic and proved [12, 11] the sanity checking property that, if $\mathsf{ClockFree}(r)$ and $\mathsf{ClockFree}(f)$ mean that r and f are unclocked, then:

$$\vdash\ \forall r\ w.\ \mathsf{ClockFree}(r) \Rightarrow (w \models^{\mathsf{T}} r = w \models r)$$
$$\vdash\ \forall f\ w.\ \mathsf{ClockFree}(f) \Rightarrow (w \models^{\mathsf{T}} f = w \models f)$$

We have also proved using the HOL system that:

$$\vdash\ \forall r\ w.\ w \models^{\mathsf{T}} r\ =\ w \models \mathcal{T}^{\mathsf{T}}(r)$$
$$\vdash\ \forall f\ w.\ w \models^{\mathsf{T}} f\ =\ w \models \mathcal{T}^{\mathsf{T}}(f)$$

which allows us to evaluate the semantics of any construct by first applying these equations and then using the unclocked semantics.

The definition of $\mathcal{T}^c(r)$ and $\mathcal{T}^c(f)$ is by recursion over the structure of SERE r and formula f. For SEREs:

$$\begin{array}{l}
(\mathcal{T}^c(b) = (\neg c[*]; c \wedge b)) \wedge \\
(\mathcal{T}^c(r_1; r_2) = \mathcal{T}^c(r_1); \mathcal{T}^c(r_2)) \wedge \\
(\mathcal{T}^c(r_1 : r_2) = \mathcal{T}^c(r_1) : \mathcal{T}^c(r_2)) \wedge \\
(\mathcal{T}^c(\{r_1\} \mid \{r_2\}) = \{\mathcal{T}^c(r_1)\} \mid \{\mathcal{T}^c(r_2)\}) \wedge \\
(\mathcal{T}^c(\{r_1\}\&\&\{r_2\}) = \{\mathcal{T}^c(r_1)\}\&\&\{\mathcal{T}^c(r_2)\}) \wedge \\
(\mathcal{T}^c(r[*]) = \mathcal{T}^c(r)[*]) \wedge \\
(\mathcal{T}^c(r@c_1) = (\neg c_1[*]; c_1 : \mathcal{T}^{c_1}(r)))
\end{array}$$

and for formulas:

$$\begin{array}{l}
(\mathcal{T}^c(b) = b) \wedge \\
(\mathcal{T}^c(\neg f) = \neg \mathcal{T}^c(f)) \wedge \\
(\mathcal{T}^c(f_1 \wedge f_2) = \mathcal{T}^c(f_1) \wedge \mathcal{T}^c(f_2)) \wedge \\
(\mathcal{T}^c(X!\, f) = X!\ ([\neg c\ U\ (c \wedge \mathcal{T}^c(f))])) \wedge \\
(\mathcal{T}^c([f_1\ U\ f_2]) = [(c \Rightarrow \mathcal{T}^c(f_1))\ U\ (c \wedge \mathcal{T}^c(f_2))]) \wedge \\
(\mathcal{T}^c(\{r\}(f)) = \{\mathcal{T}^c(r)\}([\neg c\ U\ (c \wedge \mathcal{T}^c(f))])) \wedge \\
(\mathcal{T}^c(\{r_1\} \mapsto \{r_2\}!) = \{\mathcal{T}^c(r_1)\} \mapsto \{\mathcal{T}^c(r_2)\}!) \wedge \\
(\mathcal{T}^c(\{r_1\} \mapsto \{r_2\}) = \{\mathcal{T}^c(r_1)\} \mapsto \{\mathcal{T}^c(r_2)\}) \wedge \\
(\mathcal{T}^c(f\ \mathit{abort}\ b) = \mathcal{T}^c(f)\ \mathit{abort}\ (c \wedge b)) \wedge \\
(\mathcal{T}^c(f@c_1!) = [\neg c_1\ U\ (c_1 \wedge \mathcal{T}^{c_1}(f))])
\end{array}$$

3 Executing the formal semantics

The HOL system has an ML function `EVAL` [5] which when applied to a term t proves a theorem $\vdash t = t'$, where t' is the result of evaluating t. `EVAL` performs call-by-value order rewriting efficiently using logic definitions that are in force in the context in which it is invoked. It can also invoke equations and decision procedures that have been explicitly added to the context.

3.1 Executing the clock removal rewrites

The semantics of a formula f with respect to a path w is $w \models^{\mathsf{T}} f$. The first step in evaluating $w \models^{\mathsf{T}} f$ is to rewrite with the equations:

$$\vdash\ \forall r\ w.\ w \models^{\mathsf{T}} r\ =\ w \models \mathcal{T}^{\mathsf{T}}(r) \text{ and } \vdash\ \forall f\ w.\ w \models^{\mathsf{T}} f = w \models \mathcal{T}^{\mathsf{T}}(f)$$

The next step is to execute the definition of $\mathcal{T}^{\mathsf{T}}$, and the final step is to evaluate the unclocked semantics (Section 3.2).

The clocking removal rewrites are directly executable, but the results are complicated. For example `EVAL` $(\mathcal{T}^{c}(\mathsf{T}[*];\{\neg rq@c_1\}\&\&\{ak@c_2\};rq@c_1))$ evaluates to the almost completely incomprehensible theorem:

$$\begin{array}{l}\vdash\ \mathcal{T}^{c}(\mathsf{T}[*];\{\neg rq@c_1\}\&\&\{ak@c_2\};rq@c_1) = \\ \quad \neg c[*];c\wedge\mathsf{T}[*];\{\neg c_1[*];c_1:\neg c_1[*];c_1\wedge\neg rq\}\&\&\{\neg c_2[*]; \\ \quad c_2:\neg c_2[*];c_2\wedge ak\};\neg c_1[*];c_1:\neg c_1[*];c_1\wedge rq\end{array}$$

This illustrates how much more natural and high-level are properties expressed using the $@c$ clocking construct. Note also that $c_1 : \neg c_1[*]$ is equivalent to c_1, which shows the need to perform peephole optimisations on the output of naive evaluation with the rewrites. Executing the rewrites for formulas typically produces even more incomprehensible results than with SEREs! For example, consider the following (the operator *before* is defined in Section 3.3):

$$\begin{array}{l}\vdash\ \mathcal{T}^{c}(\{\mathsf{T}[*];\neg ak_1;ak_1;\mathsf{T}\}(\neg ak_2\wedge X!\ (ak_2)\ \mathit{before}\ \neg ak_1\wedge X!\ (ak_1))) = \\ \quad \{\neg c[*];c\wedge\mathsf{T}[*];\neg c[*];c\wedge\neg ak_1;\neg c[*];c\wedge ak_1; \\ \quad \neg c[*];c\wedge\mathsf{T}\}([\neg c\ U\ c\wedge\neg\neg[\neg\neg\neg c\wedge\neg\neg\neg ak_1\wedge X!\ ([\neg c\ U\ c\wedge ak_1])\ U\ c\wedge\neg ak_2\wedge \\ \quad X!\ ([\neg c\ U\ c\wedge ak_2])\wedge\neg\neg ak_1\wedge X!\ ([\neg c\ U\ c\wedge ak_1])]\wedge\neg\neg[\neg\neg\neg c\wedge\neg\mathsf{T}\ U\ c\wedge \\ \quad \neg\neg\neg ak_1\wedge X!\ ([\neg c\ U\ c\wedge ak_1])]])\end{array}$$

Just looking at this suggests that boolean simplifications should be applied to the result of naive evaluation. Simple evaluation like this can provide a useful tool development aid, as concrete examples may provide insight into the semantics of clocking that is not immediately apparent from the general semantic definitions.

3.2 Executing the unclocked formula semantics

In some cases $w \models f$ can be executed directly, in other cases we have to first transform it into a different form.

Boolean expressions
The semantics of boolean expressions can be directly evaluated. For example, $[s]w \models^{\mathsf{T}} a \wedge b$ evaluates to $a \in s \wedge b \in s$ (if s were an explicit set rather than a variable then `EVAL` could reduce this further).

Negations $\neg f$, conjunctions $f_1 \wedge f_2$ and next-state $X!\,f$
To evaluate formulas, first note that $w \models \neg f$, $w \models f_1 \wedge f_2$ and $w \models X!\,f$ can be rewritten directly using the semantics. For example, here are the results of invoking `EVAL` on $p \wedge X!\,f$ with three increasingly specific paths (in each case `EVAL` is applied to the term on the left hand side of the equation, and generates a theorem showing the evaluation of this term):

$$\vdash\ w \models p \wedge X!\,f \ = \ (|w| > 0 \wedge w_0 \models p) \wedge |w| > 1 \wedge (w^1) \models f$$

$$\vdash\ ([s_0]w) \models p \wedge X!\,f \ = \ s_0 \models p \wedge |w| + 1 > 1 \wedge w \models f$$

$$\vdash\ s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 \models p \wedge X!\,f \ = \ s_0 \models p \ \wedge \ s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 \models f$$

These illustrate symbolic evaluation: when laws apply they are used to reduce a term, but if no laws are applicable then the term is left unevaluated: $|w|+1 > 0$ can be evaluated, since `EVAL` has been told $\vdash\ \forall n.\ n+1 > 0 = \mathsf{T}$, but $|w|+1 > 1$ cannot be evaluated for an arbitrary variable w, but the more specific term $|s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9| + 1 > 1$ can be evaluated: even though the states s_i are left as variables, since the path has length 10, which is greater than 0. With a fully concrete path the truth of the formula is completely determined. To display concrete examples we write $\{\ldots\}\{\ldots\}\cdots\{\ldots\} \models f$ where $\{\ldots\}$ are sets of atomic propositions representing states. Note that in such examples braces are set brackets, not part of SERE syntax. For example:

$$\vdash\ \{a\}\{a,b\}\{b\} \models a \wedge X!\,(b) \ = \ \mathsf{T}$$

Until formulas $[f_1\ U\ f_2]$
The semantics of the until-construct is:

$$w \models [f_1\ U\ f_2] \ = \ \exists k \in (0\ ..\ |w|).\ w^k \models f_2 \wedge \forall j \in (0\ ..\ k).\ w^j \models f_1$$

which cannot be directly executed, but there is a standard recursive version of this definition that can easily be proved as a theorem and is directly executable:

$$\vdash\ w \models [f_1\ U\ f_2] \ = \ |w| > 0 \wedge (w \models f_2 \vee w \models f_1 \wedge w^1 \models [f_1\ U\ f_2])$$

The following example is from the Reference Manual [3, Example 2, page 45].

```
time  0  1  2  3  4  5  6  7  8  9
----------------------------------
clk1  0  1  0  1  0  1  0  1  0  1
a     0  0  0  1  1  1  0  0  0  0
b     0  0  0  0  0  1  0  1  1  0
c     1  0  0  0  0  1  1  0  0  0
clk2  1  0  0  1  0  0  1  0  0  1
```

Define *w1* to be this path, namely:

w1 = `{c,clk2}{clk1}{}{clk1,a,clk2}{a}{clk1,a,b,c}{c,clk2}`
`{c,clk2}{clk1,b}{b}{clk1,clk2}`

Recall that weak clocking is defined by: $f@c \ = \ \neg(\neg f@c!)$. After making this definition we can evaluate examples like $(w1^i) \models^{\mathsf{T}} (c \wedge X!\ ([a\ U\ b]))@clk_1$ and $(w1^i) \models^{\mathsf{T}} (c \wedge X!\ (([a\ U\ b])@clk_1))@clk_2$ for $0 \leq i < |w1|$ and confirm that the first is true only when i is 4 or 5, and the second only when i is 0. The semantics of multiple clocking is subtle (clocks do not accumulate: an inner clock ignores an outer one) and is still under discussion and may change. Our tools facilitate experiments on concrete scenarios to gain insight into the current semantics.

Suffix implication $\{r\}(f)$
Suffix implication formulas $\{r\}(f)$ are executed by generating a matcher for r and then invoking `EVAL` on f whenever a match is found. In detail, the SERE r is first lifted to an element of a HOL theory of regular expressions (based on Nipkow's Isabelle work [13], but with many details adjusted for PSL), and then a proof procedure lazily constructs the state set, accepting states and transition table of an equivalent DFA. This DFA is run along a finite trace w, and whenever it enters an accepting state `EVAL` is used to check $x \models f$ on the remaining trace x. The constants that do this lifting ($\mathsf{sere2regexp}$) and DFA execution (acheck) are defined to be executed efficiently in the logic, but the following theorem shows that they also preserve the semantics of the original suffix implication formula:

$$\vdash\ \forall w\ r\ f.\ \mathsf{ClockFree}(r) \Rightarrow (w \models \{r\}(f) = \mathsf{acheck}\ (\mathsf{sere2regexp}\ r)\ (\lambda x.\ x \models f)\ w)$$

Strong suffix implications $\{r_1\} \mapsto \{r_2\}!$
Strong implications $\{r_1\} \mapsto \{r_2\}!$ are reduced to suffix implications by:[1]

$$\vdash\ \forall w\ r_1\ r_2.\ w \models \{r_1\} \mapsto \{r_2\}! = w \models \{r_1\}(\neg\{r_2\}(\mathsf{F}))$$

Weak suffix implications $\{r_1\} \mapsto \{r_2\}$
Weak implications are executed by, if necessary, performing a reachability calculation inside HOL. We add a Prefix operator[2] to the HOL regular expression theory, with the semantics that $\mathsf{Prefix}(r)$ matches a word w if it can be extended by w' such that r matches ww'. We can now use our generic lifting and DFA execution constants to execute weak implication, and the following theorem guarantees that the semantics of the original formula are preserved:

$$\begin{array}{l}\vdash\ \forall w\ r_1\ r_2.\\ \quad \mathsf{ClockFree}(r_1) \wedge \mathsf{ClockFree}(r_2) \Rightarrow\\ \quad\ w \models \{r_1\} \mapsto \{r_2\} =\\ \qquad\quad \mathsf{acheck}\ (\mathsf{sere2regexp}\ r_1)\\ \qquad\qquad\qquad (\lambda x.\ x \models \neg\{r_2\}(\mathsf{F}) \vee \mathsf{amatch}\ (\mathsf{Prefix}\ (\mathsf{sere2regexp}\ r_2))\ x)\ w\end{array}$$

The amatch constant checks whether a regular expression matches a word, by building an equivalent DFA, executing it along the word, and testing whether it is in an accepting state at the end. If the regular expression is $\mathsf{Prefix}(r)$, then the state s is accepting precisely when it is possible to reach an accepting state from s on the transition graph of (the DFA corresponding to) r. To implement this, we defined a version of Dijkstra's reachability algorithm, and proved it correct [6].

To summarise, we execute $w \models \{r_1\} \mapsto \{r_2\}$ solely by deductions in the logical kernel. We first use the above theorem to reduce the problem to executing a DFA. This involves performing many on-the-fly deductions to evaluate transitions and accepting states. The Prefix operator is the most complex of these on-the-fly deductions, requiring a reachability calculation on the transition graph. This reachability calculation can be reduced to an instance of Dijkstra's algorithm, but to make that step we need the correctness proof of the

[1] This equivalence was first observed by Dana Fisman (private communication).
[2] The prefix operators used for weak implication (Prefix) and abort ($\mathsf{FormPrefix}$) are based on an idea from Dana Fisman (private communication).

algorithm. The end result of all this deduction is a HOL theorem of the form $\vdash\ (w \models \{r_1\} \mapsto \{r_2\}) = b$, where b is either T, F, or something more complex if the original term contained variables.

Aborts $f\ abort\ b$

We currently do not have a fully general method of executing $w \models f\ abort\ b$, but evaluation in some cases is possible. First define a formula prefix function $\mathsf{FormPrefix}$ and an auxiliary function $\mathsf{AbortAux}$.

$$\begin{aligned}\mathsf{FormPrefix}\ w\ f &= \exists w'. ww' \models f\\ \mathsf{AbortAux}\ w\ f\ b\ n &= \exists j \in n\ ..\ |w|\,.w^j \models b \wedge \mathsf{FormPrefix}\ w^{0,j-1}\ f\end{aligned}$$

then it is easy to prove:

$$\begin{aligned}&\vdash\ w \models f\ abort\ b\ =\ w \models f \vee w \models b \vee \mathsf{AbortAux}\ w\ f\ b\ 1\\ &\vdash\ \mathsf{AbortAux}\ w\ f\ b\ n =\\ &\qquad n < |w| \wedge (w^n \models b \wedge \mathsf{FormPrefix}\ w^{0,n-1}\ f \vee \mathsf{AbortAux}\ w\ f\ b\ n+1)\end{aligned}$$

and adding these to the rewrites used by `EVAL` enables $f\ abort\ b$ to be executed in the trivial cases when $w \models f$ or $w \models b$ evaluate to true. For a non-trivial concrete example, consider the following (c.f. [14, Fig. 8, page 22]):

time	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
start	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
req	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
ack	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
interrupt	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

this corresponds to the finite path $w2$ where

$w2 =$ `{}{start}{}{}{req}{}{}{}{}{}{interrupt}{}{}{}{}{}`

If we define:

$$\begin{aligned}&\forall f.\ F\ f = [\mathsf{T}\ U\ f], &&\forall f.\ eventually!f = F\ f\\ &\forall f.\ G\ f = \neg(F\ (\neg f)), &&\forall f.\ always\ f = G\ f\end{aligned}$$

then `EVAL` will prove:

$$\begin{aligned}\vdash\ & w2 \models always(start \rightarrow (always(req \rightarrow eventually!ack)\ abort\ interrupt))\\ &=\\ &\mathsf{FormPrefix}\ (\{start\}\{\}\{\}\{req\}\{\}\{\}\{\}\{\}\{\})\ \neg[\mathsf{T}\ \mathsf{U}\ \neg\neg\neg\neg req \wedge \neg[\mathsf{T}\ \mathsf{U}\ ack]]\end{aligned}$$

The right hand side of this equation is true if path $\{start\}\{\}\{\}\{req\}\{\}\{\}\{\}\{\}\{\}$ can be extended to make $(\neg[\mathsf{T}\ U\ \neg\neg\neg\neg req \wedge \neg[\mathsf{T}\ U\ ack]])$ true. In this particular case it is sufficient to only consider extensions either by the empty path or by a singleton path consisting of one state of the form $\{x\}$. The following easily proved theorem says that $w \models f$ or $\exists x.(w[\{x\}]) \models f$ is sufficient.

$$\vdash\ \mathsf{FormPrefix}\ w\ f = w \models f \vee (\exists x.(w[\{x\}]) \models f) \vee \mathsf{FormPrefix}\ w\ f$$

For the example above, adding this to the equations used by `EVAL` results in a term $\exists x.\neg(\neg(ack = x) \vee (req = x) \wedge \neg(ack = x))$ being generated. `EVAL` can be programed to invoke a decision procedure on such terms. This is an ad hoc partial solution. We hope eventually to make our implementation complete.

3.3 More examples

The first example below illustrates the utility of having an automatic semantics calculator. The second example shows how such a calculator can be used to analyse behaviours captured from simulation.

3.3.1 An example from an Accellera online discussion

A recent online discussion [9, 10] concerned the intervals for which the SERE $(((a;b))@clk_1;c)@clk_2$ "hold tightly" within the behaviour:

```
time  0  1  2  3  4  5  6  7
----------------------------
clk1  0  1  0  1  0  1  0  1
a     0  1  1  0  0  0  0  0
b     0  0  0  1  0  0  0  0
c     0  0  0  0  1  0  1  0
clk2  1  0  0  1  0  0  1  0
```

The Reference Manual introduces the terminology r "holds tightly" for w if and only if $w \overset{\mathsf{T}}{\models} r$. To understand this example, note that clocks don't accumulate: only the current, i.e. innermost, one is used to sample the path. To analyse this example, a simple ML function can easily be written that evaluates a SERE on all sub-intervals of a path and returns the results that correspond to intervals for which the SERE holds. Using this we can analyse the example and deduce that the only interval where the SERE holds tightly is:

$$\vdash \ \{clk_2\}\{clk_1, a\}\{a\}\{clk_1, b, clk_2\}\{c\}\{clk_1\}\{c, clk_2\} \overset{\mathsf{T}}{\models} a; b@clk_1; c@clk_2 = \mathsf{T}$$

This resolves the discussion in favour of the Manual ([10] is correct, [9] is wrong). Note that the other examples in the Manual can be (and have been) automatically checked. The fact that there is ongoing discussion about properties as simple as this suggests that our semantics calculator might be a useful tool for property writers.

3.3.2 An example from the FoCs Manual

Evaluation in HOL is nearly instantaneous for examples of the scale above. Whilst we would not claim our evaluator can handle 'industrial scale' problems, it can be applied to significantly more complex examples. In the IBM FoCs Manual there is a Sender-Buffer-Receiver in which the Sender (S) communicates with the buffer (B) using four-phase handshakes with request signal $StoB\_REQ$ and acknowledgement $BtoS\_ACK$, and the Buffer communicates with the Receiver (R) with a four-phase handshake with request signal $BtoR\_REQ$ and acknowledgement $RtoB\_ACK$.

We can define in HOL a function $\mathsf{FourPhase}$ such that $\mathsf{FourPhase}\ req\ ack$ is true if signals req and ack satisfy properties required of a four-phase handshake. First define:

$$\forall r.\ never(r) = \{\mathsf{T}[*]; r\} \mapsto \{\mathsf{F}\}$$

then define:

$$\begin{array}{l}\mathsf{FourPhase}\ req\ ack = \\ \quad never(\mathsf{T}[*]; \neg req \wedge ack; req) \ \wedge \ never(\mathsf{T}[*]; req \wedge \neg ack; \neg req) \wedge \\ \quad never(\mathsf{T}[*]; \neg ack \wedge \neg req; ack) \ \wedge never(\mathsf{T}[*]; ack \wedge req; \neg ack)\end{array}$$

Definitions like FourPhase in HOL are analogous to definitions of verification units (`vunits`) in PSL.

We have written a Verilog model to generate paths. If *SimRun* is a 700 state Verilog generated path, our tool currently takes about a couple of minutes on a 1GHz machine to evaluate: $SimRun \models^{\mathsf{T}}$ FourPhase $StoB\_REQ$ $BtoS\_ACK$ and $SimRun \models^{\mathsf{T}}$ FourPhase $BtoR\_REQ$ $RtoB\_ACK$. Notice that both *never* and FourPhase have an initial $\mathsf{T}[*]$. If we remove the occurrences of $\mathsf{T}[*]$ in FourPhase then the checking is more than twice as fast. If we augmented the rewrites used by `EVAL` to include:

$$\vdash\ \forall w\ r_1\ r_2\ r_3.\ w \models (r_1; r_2); r_3\ =\ w \models r_1; (r_2; r_3)$$
$$\vdash\ \forall w\ r.\ w \models r[*]; r[*]\ =\ w \models r[*]$$

then this optimisation could be made to happen automatically.

If, using the definition of $G\ f$ given earlier, we define:

$$[f_1\ W\ f_2] = [f_1\ U\ f_2] \vee G\ f_1, \qquad f_1\ \mathit{before}\ f_2 = [\neg f_2\ W\ f_1 \wedge \neg f_2]$$

then AckInterleave ack_1 ack_2 defined below states that ack_2 is asserted between any two ack_1 assertions:

$$\begin{array}{l}\mathsf{AckInterleave}\ ack_1\ ack_2 = \\ \quad \{(\mathsf{T}[*]; \neg ack_1; ack_1)\}(\neg ack_2 \wedge X!\ (ack_2)\ \mathit{before}\ \neg ack_1 \wedge X!\ (ack_1))\end{array}$$

Checking that the conjunction below evaluates to T takes about 5 minutes.

$$\begin{array}{l} SimRun \models^{\mathsf{T}} \mathsf{AckInterleave}\ BtoS\_ACK\ RtoB\_ACK\ \wedge \\ SimRun \models^{\mathsf{T}} \mathsf{AckInterleave}\ RtoB\_ACK\ BtoS\_ACK\end{array}$$

This corresponds to the `vunit ack_interleaving` in the FoCs Manual example.

4 Compiling the formal semantics

In the last section we saw how to execute the formal semantics by deduction in the theorem prover. In particular, SEREs are executed by constructing a provably equivalent DFA. In the same way, some PSL formulas are equivalent to DFAs, where a violation of the formula corresponds to the DFA entering an accepting state. In this section, we show how to safely compile a subset of such PSL formulas as 'checker modules' in a HDL. An off-the-shelf simulation tool is then used to simulate the circuit together with the checker, and any violations of the property are detected and reported to the user.

To illustrate the operation of the compiler, we will use part of the FourPhase property (introduced in Section 3.3).

$$\mathit{never}(\neg StoB\_REQ \wedge BtoS\_ACK; StoB\_REQ)$$

This says that whenever $StoB\_REQ$ is low and $BtoS\_ACK$ is high, it is never the case that $StoB\_REQ$ will go high before $BtoS\_ACK$ goes low. By the definition of *never* (also in Section 3.3), this property holds if and only if the following SERE does *not* hold for any initial segment of the trace:

$$\mathsf{T}[*]; \neg StoB\_REQ \wedge BtoS\_ACK; StoB\_REQ$$

If we convert this SERE to an equivalent DFA, it is easy to check whether it accepts any initial segment of a trace. We simply advance the DFA along the trace according to its transition function, and if it ever reaches an accepting state we report that the *never* property has been violated.

To summarise, compiling the property $never(r)$ reduces to generating an equivalent DFA to the SERE $\mathsf{T}[*]; r$, and replacing accepting states with an error message reporting that the property has been violated.

Let us now look more closely at the compilation process, to see how the semantics are preserved. We begin with the PSL formula $never(r)$. We convert the SERE $\mathsf{T}[*]; r$ to an element of the HOL regular expression theory, and then to a DFA with a set of states, a subset of accepting states and a transition table. We intend to simulate this DFA concurrently with a circuit, and report an error whenever the DFA enters an accepting state. We can consider the circuit simulation to be producing an infinite trace, and the DFA effectively run on all initial segments of this. The following theorem shows that this mode of operation preserves the semantics of the original PSL formula $never(r)$:

$$\begin{aligned}\vdash\ & \forall r\ w.\ \mathsf{ClockFree}(r) \land (|w| = \infty) \Rightarrow \\ & \qquad (w \models never(r) = \forall n.\ \neg\mathsf{amatch}\ (\mathsf{sere2regexp}\ (\mathsf{T}[*]; r))(w^{0,n}))\end{aligned}$$

The next step is the extraction of the DFA from HOL to an ML data-type, ready for a compiler back end to output code for a particular HDL. We use proof as much as possible in this function, because it increases our confidence in the correctness of the extracted DFA while incurring relatively little cost. The ML function that performs the extraction takes as input a list of atomic propositions and a regular expression, and returns for each *reachable* state of the DFA: (i) an integer state identifier, and the HOL term that represents the state, (ii) a boolean that is true for accepting states, and a HOL theorem proving this, and (iii) a 'condition' data-type that encodes a series of tests on the truth value of atomic propositions followed by a transition to a new state, with HOL theorems proving the conditional transitions correct.

Shown below is the ML output from applying the DFA extraction function to our example: $\mathtt{R} := \mathsf{sere2regexp}\ (\mathsf{T}[*]; \neg StoB\_REQ \land BtoS\_ACK; StoB\_REQ)$.

```
[(0, `[6]`, (false, |- eval_accepts R [6] = F),
  Branch("StoB_REQ",
    Leaf(1, |- !s. StoB_REQ IN s ==> (eval_transitions R [6] s = [4])),
    Branch("BtoS_ACK",
      Leaf(2, |- !s.
                  ~(StoB_REQ IN s) /\ BtoS_ACK IN s ==>
                  (eval_transitions R [6] s = [2; 4])),
      Leaf(1, |- !s.
                  ~(StoB_REQ IN s) /\ ~(BtoS_ACK IN s) ==>
                  (eval_transitions R [6] s = [4]))))),
  (1, `[4]`, (false, |- eval_accepts R [4] = F), ...),
  (2, `[2; 4]`, (false, |- eval_accepts R [2; 4] = F), ...),
  (3, `[0; 4]`, (true, |- eval_accepts R [0; 4] = T), ...)]
```

For reasons of space only the transition function for state 0 (the initial state) is shown. The term representing this state is `[6]`[3], the `false` indicates that this state is not accepting, and is followed by a theorem proving this.

[3] The values of the HOL terms representing states are an artifact of the DFA subset construction, and should be treated as arbitrary terms.

The condition first tests the atomic proposition StoB_REQ, and if true moves to state 1 (which as we see is represented in HOL as [4]). The conditional theorem at this leaf reflects this transition.

From this language independent description of a DFA, it is a simple matter to generate versions in a HDL. We have implemented a pretty-printer for Verilog syntax. The resulting Verilog module for our example property is shown in Fig. 1, and it has correctly reported errors during simulations of a buggy buffer circuit.

```
module Checker (StoB_REQ, BtoS_ACK, BtoR_REQ, RtoB_ACK);

input       StoB_REQ, BtoS_ACK, BtoR_REQ, RtoB_ACK;
reg   [1:0] state;

initial state = 0;

always @ (StoB_REQ or BtoS_ACK or BtoR_REQ or RtoB_ACK)
begin
 $display ("Checker: state = %0d", state);
 case (state)
  0: if (StoB_REQ) state = 1; else if (BtoS_ACK) state = 2; else state = 1;
  1: if (StoB_REQ) state = 1; else if (BtoS_ACK) state = 2; else state = 1;
  2: if (StoB_REQ) state = 3; else if (BtoS_ACK) state = 2; else state = 1;
  3: begin $display ("Checker: property violated!"); $finish; end
  default: begin $display ("Checker: unknown state"); $finish; end
 endcase
end

endmodule
```

Fig. 1. The Verilog state machine for the example property.

5 Conclusions and future work

The main point of this paper is that a formal semantics is not just documentation. Current theorem provers are powerful enough to be programmed to execute semantics in interesting ways, though a major challenge is to engineer the deductions to be fast enough to be useful. We have illustrated this with two prototype tools. The first one could be useful for property developers and teachers and learners of PSL. The second one illustrates a novel way of implementing an EDA tool that guarantees conformance to the standard. We think such semantics-based tools could eventually be made efficient enough for industrial scale use, but one needs to choose applications where semantic accuracy is more critical than performance. The times (minutes) quoted in Section 3.3 will not impress members of the model checking community, but this doesn't necessarily mean they are unacceptable, given the correct-by-construction benefits of the implementation method.

The work described here illustrates a convergence of computation and deduction, in which the execution of theorem proving strategies becomes a powerful method of implementation. We plan to extend, package and ruggedise our prototypes into standalone tools that automatically invoke HOL (currently they are invoked from HOL via ML functions). The interpreter is complete excepts for aborts, but the checker only handles a subset of formulas. Our goal is to cover the whole of PSL.

6 Acknowledgements

Thanks to Cindy Eisner, Dana Fisman and Hasan Amjad for help with our research, and Keith Wansbrough for help preparing the paper. Additional thanks to Cindy Eisner for comments on an earlier version of this paper.

References

1. Yael Abarbanel, Ilan Beer, Leonid Gluhovsky, Sharon Keidar, and Yaron Wolfsthal. FoCs: Automatic Generation of Simulation Checkers from Formal Specifications. In *Computer Aided Verification*, pages 538–542, 2000. www.haifa.il.ibm.com/projects/verification/RB_Homepage/ps/checkers.ps.
2. Accellera home page: www.accellera.org.
3. Accellera Property Specification Lanuage Reference Manual, Version 1.0. www.eda.org/vfv/docs/psl_lrm-1.0.pdf.
4. H. Amjad. Programming a symbolic model checker in a fully expansive theorem prover. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logic*, LNCS. Springer-Verlag, 2003. To appear.
5. Bruno Barras. Programming and computing in HOL. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, number 1869 in Lecture Notes in Computer Science, pages 17–37. Springer-Verlag, 2000.
6. Thomas H. Cormen, Charles Eric Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, Cambridge, Massachusetts, 1990.
7. L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham. The prosper toolkit. In S. Graf and M. Schwartbach, editors, *Tools and Algorithms for Constructing Systems (TACAS 2000)*, number 1785 in Lecture Notes in Computer Science, pages 78–92. Springer-Verlag, 2000.
8. The Accellera Formal Property Language Technical Committee home page: www.eda.org/vfv.
9. Comment on Ex 2, p 34, PSL RM v1.0: /www.eda.org/vfv/hm/1017.html.
10. Reply to Comment on Ex 2, p 34, PSL RM v1.0: /www.eda.org/vfv/hm/1019.html.
11. Michael J. C. Gordon. Validating the PSL/Sugar semantics using automated reasoning. *Formal Aspects of Computing*. Special issue on Semantic Foundations of Engineering Design Languages (to appear).
12. Michael J. C. Gordon. Using HOL to study Sugar 2.0 semantics. In Victor A. Carreño, Cesar A. Muñoz, and Sofieǹe Tahar, editors, *Track B Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics, TPHOLs2002*, volume CP-2002-211736 of *NASA Conference Proceedings*, pages 87–100, 2002. http://shemesh.larc.nasa.gov/tphols2002/proceedings.html.
13. Tobias Nipkow. Verified lexical analysis. In Jim Grundy and Malcolm C. Newey, editors, *Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'98, Canberra, Australia*, volume 1479 of *LNCS*. Springer, 1998.
14. Proposal Presented to the Accellera Formal Verification Technical Committee `www.haifa.il.ibm.com/projects/verification/sugar/Sugar_2.0_Accellera.ps`.