Executing the formal semantics of the Accellera Property Specification Language
— joint work with Joe Hurd & Konrad Slind —

Standard practice: generate tools from formal syntax

- Parser generator
- Input a grammar
- Apply theory of formal languages
- Generate a parser
Idea of this talk: generate tools from formal semantics

- Input ‘golden’ semantics from LRM
- Perform mechanised proof
- Generate tools
Goals and non-Goals

- Goal is to show formal semantics is not just documentation
  - can run the Language Reference Manual (LRM)

- Correctness primary, efficiency secondary
  - but need sufficient efficiency!

- Programming methodology, not new verification algorithms
  - EDA tools with theorem prover inside (c.f. PROSPER)
Accellera’s PSL (formerly IBM’s Sugar 2.0)

- PSL is a property specification language combining
  - boolean expressions (Verilog syntax)
  - patterns (Sequential Extended Regular Expressions SEREs)
  - LTL formulas (Foundation language FL)
  - CTL formulas (Optional Branching Extension OBE)

- Designed both for model checking and simulation testbenches

- Intended to be the industry standard
Generating PSL tools

Official semantics of PSL

HOL 4 THEOREM PROVER

TOOL1: evaluate FL properties on a specific path

TOOL2: compile properties to HDL checkers (idea from FoCs)

TOOL3: check OBE properties against a model (Amjad’s PhD)
Tools use standard algorithms

► TOOL1: semantic calculator
  • match regexps using automata; evaluate formulas recursively
  • automata constructed and executed by proof inside HOL

► TOOL2: checker compiler
  • compile regexps to automata, then ‘pretty print’ to HDL (Verilog)
  • treatment of formulas incomplete and *ad hoc*

► TOOL3: symbolic model checker
  • classical McMillan-style $\mu$-calculus checker
  • uses BDD representation judgements to link HOL terms to BDDs
  • see Gordon (TPHOLs2001), Amjad (TPHOLs2003)

► **No** new algorithms, but **maybe** a new kind of logic programming
Theorem proving often associated with heroic proofs
- e.g. Gödel’s theorem (Shankar), relative consistency of AC (Paulson)

We are not doing heroic proofs, but a kind of logic programming
- computation by deduction

HOL has a relatively fast call-by-value symbolic evaluator EVAL
- by Bruno Barras using Coq technology (explicit substitutions)
- doesn’t compete with ACL2 or PVS ground evaluators (or C, C++)
- runs ARM6 microarchitecture at a few seconds per instruction
- key tool for our PSL evaluator
Executing the semantics (note: \((0 .. |w|)\) should be \([0 .. |w|)\))

- By rewriting and evaluation (PSL in red, HOL in blue):
  \[
  \vdash s_0s_1s_2s_3s_4s_5s_6s_7s_8s_9 \models p \land X! f = s_0 \models p \land s_1s_2s_3s_4s_5s_6s_7s_8s_9 \models f
  \]
  \[
  \vdash \{a\}\{a, b\}\{b\} \models a \land X! b = \top
  \]

- LRM semantics of the until-operator not directly executable
  \[
  w \models [f_1 \ U \ f_2] = \exists k \in (0 .. |w|). w^k \models f_2 \land \forall j \in (0 .. k). w^j \models f_1
  \]

- Standard reformulation makes it directly executable
  \[
  \vdash w \models [f_1 \ U \ f_2] = |w| > 0 \land (w \models f_2 \lor w \models f_1 \land w^1 \models [f_1 \ U \ f_2])
  \]

- If \(f_1, f_2\) are boolean expressions and the path is arbitrary of length 5:
  \[
  \vdash s_0s_1s_2s_3s_4 \models [b_1 \ U \ b_2] =
  \]
  \[
  s_0 \models b_2 \lor
  s_0 \models b_1 \land (s_1 \models b_2 \lor s_1 \models b_1 \land
  (s_2 \models b_2 \lor s_2 \models b_1 \land (s_3 \models b_2 \lor s_3 \models b_1 \land s_4 \models b_2)))
  \]
Matching regular expressions

- Semantics of PSL SEREs is self-explanatory

\[
\begin{align*}
(w \models b) &= (|w| = 1) \land w_0 \models b) \\
(w \models r_1; r_2) &= \exists w_1w_2. (w = w_1w_2) \land w_1 \models r_1 \land w_2 \models r_2) \\
(w \models r_1 : r_2) &= \exists w_1w_2l. (w = w_1[l]w_2) \land w_1[l] \models r_1 \land [l]w_2 \models r_2) \\
(w \models \{r_1\} | \{r_2\}) &= w \models r_1 \lor w \models r_2) \\
(w \models \{r_1\} \& \& \{r_2\}) &= w \models r_1 \land w \models r_2) \\
(w \models r[^*]) &= \exists wlist. (w = \text{Concat} wlist) \land \text{Every}(\lambda w. w \models r)wlist)
\end{align*}
\]

- Make executable by proving

\[
\vdash \ \forall w \ r. \ w \models r = \text{amatch(sere2regexp}(r))w
\]

where:

- sere2regexp converts a SERE to a HOL regular expression
- amatch is an executable matcher for regular expressions
Suffix implication \( \{r\}(f) \)

- Semantics is:
  \[
  w \models \{r\}(f) = \forall j \in (0 .. |w|). w^{0:j} \models r \Rightarrow w^j \models f
  \]

- Have defined an efficient executable function `ach` so that, for example:
  
  \[
  \text{ach} \ f [x_0; x_1; x_2; x_3] = \\
  (\text{amatch} \ r [x_0] \Rightarrow f[x_1; x_2; x_3]) \land \\
  (\text{amatch} \ r [x_0; x_1] \Rightarrow f[x_1; x_2; x_3]) \land \\
  (\text{amatch} \ r [x_0; x_1; x_2] \Rightarrow f[x_2; x_3]) \land \\
  (\text{amatch} \ r [x_0; x_1; x_2; x_3] \Rightarrow f[x_3])
  \]

  N.B. execution only costs the same as the last `amatch` call

- Then proved
  \[
  \vdash \forall w r f. w \models \{r\}(f) = \text{ach}(\text{sere2regexp}(r))(\lambda x. x \models f)w
  \]

- Rewrite with this, then execute
Other constructs: \( \{r_1\} \leftrightarrow \{r_2\}! \) and \( \{r_1\} \leftrightarrow \{r_2\} \)

- Reduce \( \{r_1\} \leftrightarrow \{r_2\}! \) to suffix implication by proving
  \[ \vdash \forall w \, r_1 \, r_2. \, w \models \{r_1\} \leftrightarrow \{r_2\}! = w \models \{r_1\}(\neg \{r_2\}(F)) \]

- Handle \( \{r_1\} \leftrightarrow \{r_2\} \) with regular expression \( \text{Prefix}(r) \) (in HOL not PSL)
  \[ \vdash \forall r \, w. \, w \models \text{Prefix}(r) = \exists w'. \, w \, w' \models r \]

- Execution of \( w \models \text{Prefix}(r) \) uses Dijkstra’s algorithm

- Have proved:
  \[ \vdash \forall w \, r_1 \, r_2. \]
  \[ w \models \{r_1\} \leftrightarrow \{r_2\} = \]
  \[ \text{acheck}(\text{sere2regexp } r_1) \]
  \[ (\lambda x. \, x \models \neg \{r_2\}(F)) \lor \text{amatch} \left( \text{Prefix} \left( \text{sere2regexp } r_2 \right) \right) \, x \) \, \text{w} \]

- Rewrite with this, then execute
Remaining formulas: aborts and clocking

Semantics of abort formulas needs a reachability algorithm
- have implemented a partial method
- awaiting new abort semantics before attempting complete solution

Clocked formulas $f@c, f@c!$ can be translated to unclocked formulas
- translation to unclocked formulas is by a recursive function
- can be directly executed
- have proved a theorem that says (roughly):

$$\forall r. \text{unclocked\_semantics}(\text{translation } r) = \text{clocked\_semantics}(r)$$
Example

PSL Reference Manual Example 2, page 45

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>clk2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Define \( w \) to be this path, so \( w \) is:

\[
{c, clk2}{clk1}{}{clk1, a, clk2}{a}{clk1, a, b, c}{c, clk2}{clk1, b}{b}{clk1, clk2}
\]

Can evaluate in SML, or via a command line wrapper

Example: to evaluate \((c \&\& \text{next!(a until b)})@clk1\) at all times in \( w \):

```
% pslcheck -all \
   -fl 'c && next!(a until b))@clk1' \
   -path '{c,clk2}{clk1}{}{clk1,a,clk2}{a}{clk1,a,b,c}{c,clk2}{clk1,b}{b}{clk1,clk2}'
>> true at times 4,5,10
```
Uses of TOOL1 (calculating $w \models^T f$ from semantics)

- Teaching and learning tool for exploring semantics
- Checking one has the right property before using it in verification
- Post simulation analysis (path is generated by simulator)
  - compare with “TransEDA VN-Property” property checker and analyzer
  - our tools much slower – but not necessary too slow!
  - guaranteed PSL compliant by construction: golden reference
TOOL2: Compile the semantics to checkers

- Idea pinched from IBM FoCs project
- A defined operator: $\forall r. \text{never}(r) = \{ T[*]; r \} \mapsto \{ F \}$
- Example property: $\text{never}(\neg \text{StoB_REQ} \land \text{BtoS_ACK}; \text{StoB_REQ})$
- Use semantics to generate a Verilog checker

```verilog
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
```
Example of how the checker works and is justified

- The following theorem is first proved

\[ \vdash |w| = \infty \implies w \models never(r) = \forall n. \neg \text{amatch (sere2regexp } T[\ast]; r)(w^{0,n}) \]

- Thus there’s an error if \( \text{amatch (sere2regexp } T[\ast]; r)(w^{0,n}) \) is ever true

- Generate a DFA from \( \text{sere2regexp } T[\ast]; r \)

- So far everything is by proof, so correct by construction

- Final step is to pretty print checker into HDL (Verilog)
  - this may introduce errors
  - no formal semantics of Verilog :-(

- Only have ‘proof of concept’ for checkers: more work to cover all formulas
Conclusions

- Two tools: semantic calculator and checker generator
- Correct by construction
- More work needed (especially for checkers)
- Illustrates new kind of logic programming using a theorem prover
  - prototyping standards compliant tools
  - theorem proving is slow ................. but not necessarily too slow
  - maybe OK for some industrial strength performance-non-critical tools

THE END