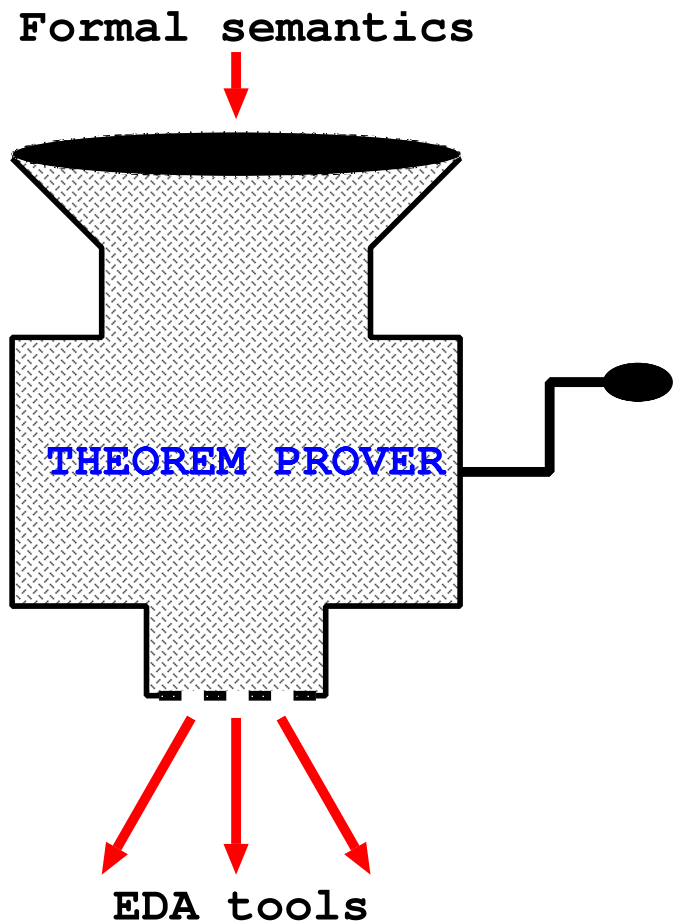
Executing the formal semantics of the Accellera Property Specification Language

— joint work with Joe Hurd & Konrad Slind —



- Input 'golden' semantics from LRM
- Perform mechanised proof
- Generate tools

Goals and non-Goals

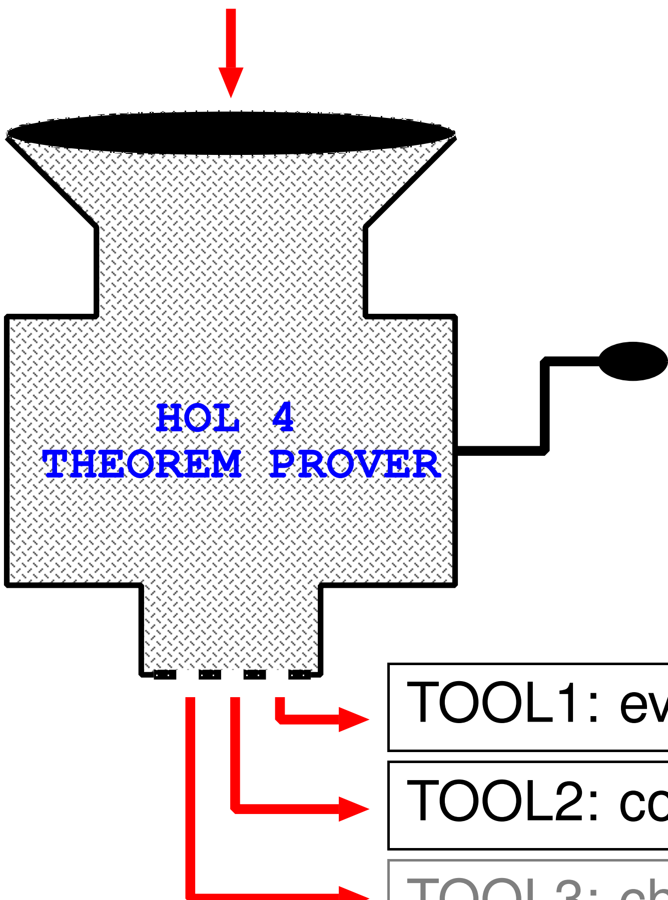
- Goal is to show formal semantics is not just documentation
  - can run the Language Reference Manual (LRM)
- Correctness primary, efficiency secondary
  - but need sufficient efficiency!
- Programming methodology, not new verification algorithms
  - EDA tools with theorem prover inside (*c.f.* PROSPER)

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



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 μ-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

Our theorem proving infrastructure (HOL)

- Standard ML infrastructure to interactively prove $\vdash t$
  - t is a term of higher order logic
  - proof is 'fully-expansive' – a sequence of primitive inference steps
- Logic is typed
  - type system supports user-defined datatypes
  - example: define types of PSL expressions, SEREs and formulas
- Contains the usual proof tools
  - simplifier (rewriter)
  - decision procedures for subsets of natural numbers, integers, reals
  - first order reasoners (inspired by Isabelle)

Heroic proofs versus 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 evalutor

Parts of semantics are directly executable

- Semantics of boolean expressions (PSL in red, HOL in blue)

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

- Fragment of semantics of formulas

  $$\begin{array}{lll}(w \models b & = & |w| > 0 \wedge w_0 \models b) \wedge \\ (w \models f_1 \wedge f_2 & = & w \models f_1 \wedge w \models f_2) \wedge \\ (w \models X!\, f & = & |w| > 1 \wedge w^1 \models f)\end{array}$$

- Examples of rewriting and evaluation:

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

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

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

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

Parts of semantics require reformulation for execution

- LRM semantics of the until-operator not directly executable

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

- Standard reformulation makes it directly executable

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

- If f_1, f_2 are boolean expressions and the path is arbitrary of length 5:

$$\begin{array}{ll} \vdash & s_0 s_1 s_2 s_3 s_4 \models [b_1 \ U \ b_2] = \\ & s_0 \models b_2 \vee \\ & s_0 \models b_1 \wedge (s_1 \models b_2 \vee s_1 \models b_1 \wedge \\ & (s_2 \models b_2 \vee s_2 \models b_1 \wedge (s_3 \models b_2 \vee s_3 \models b_1 \wedge s_4 \models b_2))) \end{array}$$

Matching regular expressions

- Semantics of PSL SEREs is self-explanatory

$$
\begin{array}{lcll}
(w \models b & = & (|w| = 1) \wedge w_0 \models b) & \wedge \\
(w \models r_1;\ r_2 & = & \exists\, w_1 w_2.\ (w = w_1 w_2) \wedge w_1 \models r_1 \wedge w_2 \models r_2) & \wedge \\
(w \models r_1 : r_2 & = & \exists\, w_1 w_2 l.\ (w = w_1[l]w_2) \wedge w_1[l] \models r_1 \wedge [l]w_2 \models r_2) & \wedge \\
(w \models \{r_1\} \mid \{r_2\} & = & w \models r_1 \vee w \models r_2) & \wedge \\
(w \models \{r_1\}\&\&\{r_2\} & = & w \models r_1 \wedge w \models r_2) & \wedge \\
(w \models r[*] & = & \exists\, wlist.\ (w = \mathsf{Concat}\ wlist) \wedge \mathsf{Every}(\lambda\, w.\ w \models r)wlist) &
\end{array}
$$

- Make executable by proving

$$\vdash \quad \forall\, w\ r.\ w \models r = \mathsf{amatch}(\mathsf{sere2regexp}(r))w$$

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 acheck so that, for example:

$$\begin{array}{l} \mathsf{acheck}\ r\ f\ [x_0;\ x_1;\ x_2;\ x_3] = \\ \quad (\mathsf{amatch}\ r\ [x_0] \Rightarrow f[x_0;\ x_1;\ x_2;\ x_3]) \wedge \\ \quad (\mathsf{amatch}\ r\ [x_0;\ x_1] \Rightarrow f[x_1;\ x_2;\ x_3]) \wedge \\ \quad (\mathsf{amatch}\ r\ [x_0;\ x_1;\ x_2] \Rightarrow f[x_2;\ x_3]) \wedge \\ \quad (\mathsf{amatch}\ r\ [x_0;\ x_1;\ x_2;\ x_3] \Rightarrow f[x_3]) \end{array}$$

- Then proved

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

- Rewrite with this, then execute

Strong suffix implication $\{r_1\} \mapsto \{r_2\}!$

- Semantics is:

$$w \models \{r_1\} \mapsto \{r_2\}! \;=\; \forall j \in [0 \,..\, |w|).w^{0,j} \models r_1 \Rightarrow \exists k \in [j \,..\, |w|).w^{j,k} \models r_2$$

- Reduced to suffix implication by proving

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

- Rewrite with this, then execute

Weak suffix implication $\{r_1\} \mapsto \{r_2\}$

- Semantics is:

$$w \models \{r_1\} \mapsto \{r_2\} \ =$$
$$\forall\, j \in [0 \,..\, |w|).$$
$$w^{0,j} \models r_1 \Rightarrow (\exists\, k \in [j \,..\, |w|).w^{j,k} \models r) \vee (\forall\, k \in [j \,..\, |w|).\, \exists\, w'.w^{j,k}w' \models r_2)$$

- Have added a special regular expression $\mathsf{Prefix}(r)$ to HOL (not to PSL)

$$\vdash\ \ \forall\, r\ w.\ w \models \mathsf{Prefix}(r)\ =\ \exists\, w'.\ w\ w' \models r$$

- Execution of $w \models \mathsf{Prefix}(r)$ uses Dijkstra's algorithm

- Have proved:

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

- Rewrite with this, then execute

Remaining formulas: aborts and clocking

- Semantics of abort formulas:

$$w \models f \ \mathit{abort}\ b \ =$$
$$w \models f \ \vee \ w \models b \ \vee \ \exists\, j \in [1 \mathinner{..} |w|).\ \exists\, w'.\ w^j \models b \wedge w^{0,j-1}w' \models f$$

  - $\exists\, w'$ 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

Clocking

- LRM defines $w \models^{c} r$ and $w \models^{c} f$ for arbitrary clock c
  - clocks c are arbitrary boolean expressions
  - top level default clock is T
- Semantics of clocked SEREs

$$w \models^{c} r@c_1 \;=\; \exists\, i \in [0\; ..\; |w|).\ w^{0,i} \models^{\mathsf{T}} \neg c_1[*];\ c_1 \wedge w^{i} \models^{c_1} r$$

- Semantics of clocked formulas

$$w \models^{c} f@c_1! \;=\; \exists\, i \in [0\; ..\; |w|).\ w^{0,i} \models^{\mathsf{T}} \neg c_1[*];\ c_1 \wedge w^{i} \models^{c_1} f$$

- Execute by rewriting with function $\mathcal{T}^{\mathsf{T}}$ and then the theorems:

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

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

Example

- PSL Reference Manual 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 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}`

- Example uses weak clocking defined by: $f\texttt{@}c \;=\; \neg(\neg f\texttt{@}c!)$

- Evaluation yields

$\vdash\;$ w$^6 \models^{\mathsf{T}} (c \wedge X!\ [a\ U\ b]\texttt{@}(clk_1 \vee clk_2))\texttt{@}(clk_1 \vee clk_2) =$ T

$\vdash\;$ w$^i \models^{\mathsf{T}} (c \wedge X!\ [a\ U\ b]\texttt{@}(clk_1 \vee clk_2))\texttt{@}(clk_1 \vee clk_2) =$ F (if $i \neq 6$)

SML convenient for scripting combinations of evaluations

- Example: use SML **map** function to generate

```
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
```

$\vdash \mathsf{w}^0 \models^{\mathsf{T}} c \wedge X!\ [a\ U\ b]@clk_1 = \mathsf{F}$

$\vdash \mathsf{w}^1 \models^{\mathsf{T}} c \wedge X!\ [a\ U\ b]@clk_1 = \mathsf{F}$

$\vdash \mathsf{w}^2 \models^{\mathsf{T}} c \wedge X!\ [a\ U\ b]@clk_1 = \mathsf{F}$

$\vdash \mathsf{w}^3 \models^{\mathsf{T}} c \wedge X!\ [a\ U\ b]@clk_1 = \mathsf{F}$

$\vdash \mathsf{w}^4 \models^{\mathsf{T}} c \wedge X!\ [a\ U\ b]@clk_1 = \mathsf{T}$

$\vdash \mathsf{w}^5 \models^{\mathsf{T}} c \wedge X!\ [a\ U\ b]@clk_1 = \mathsf{T}$

$\vdash \mathsf{w}^6 \models^{\mathsf{T}} c \wedge X!\ [a\ U\ b]@clk_1 = \mathsf{F}$

$\vdash \mathsf{w}^7 \models^{\mathsf{T}} c \wedge X!\ [a\ U\ b]@clk_1 = \mathsf{F}$

$\vdash \mathsf{w}^8 \models^{\mathsf{T}} c \wedge X!\ [a\ U\ b]@clk_1 = \mathsf{F}$

$\vdash \mathsf{w}^9 \models^{\mathsf{T}} c \wedge X!\ [a\ U\ b]@clk_1 = \mathsf{F}$

- Easy to evaluate SEREs and formulas on all subpaths of a path

Uses of TOOL1 (calculating $w \models^{\mathsf{T}} f$ 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: $\forall\, r.\ never(r) \;=\; \{\mathrm{T}[*];\ r\} \mapsto \{\mathsf{F}\}$
- Example property: $never(\neg\mathtt{StoB\_REQ} \wedge \mathtt{BtoS\_ACK};\ \mathtt{StoB\_REQ})$
- Use semantics to generate a Verilog checker

```
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

  $\vdash \ |w| = \infty \ \Rightarrow \ w \models never(r) = \forall\, n.\ \neg\mathsf{amatch}\ (\mathsf{sere2regexp}\ \mathrm{T}[*];\ r)(w^{0,n})$

- Thus there's an error if $\mathsf{amatch}\ (\mathsf{sere2regexp}\ \mathrm{T}[*];\ r)(w^{0,n})$ is ever true
- Generate a DFA from $\mathsf{sere2regexp}\ \mathrm{T}[*];\ r$
- 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
- Possible application: generate OVL checkers from PSL specifications
- .... THE END

ADDITIONAL SLIDES ON HOL

The HOL system

- Versions of the HOL system:
  1. HOL88 from Cambridge
  2. HOL90 from Calgary and Bell Labs
  3. HOL98 from Cambridge, Glasgow and Utah.
  4. HOL 4 open source project at SourceForge
- Current team ....`hol.sf.net`

Developer	Role/Position	Location
Anthony Fox	Developer	UK
Peter Homeier	Developer	USA
Hasan Amjad	Developer	UK
Joe Hurd	Developer	UK
Ken Friis Larsen	Advisor/Mentor/Consultant	Denmark
Keith Wansbrough	Developer	UK
Michael Norrish	**Project Manager**	Australia
Mike Gordon	Developer	UK
Konrad Slind	**Project Manager**	USA

- No longer managed from Cambridge

New tools (some here, some coming soon)

- New theorem proving tactics
  - ordered resolution and paramodulation for equality reasoning
  - time-sliced combinations of resolution and model elimination
- New decision procedure for full Presburger arithmetic
  - Pugh's "Omega Test"
- Improved support for emulating predicate subtypes
  - PVS is still better `:-(`
- Fully-expansive model checking
  - CTL checking as proof in representation judgement calculus
- Tools for 'boolification' to encode for BDD and SAT
  - automatically generate encoders/decoders from datatype definition
  - automatically generate bitvector versions of function definitions

Some recent or current projects

- Verification of AES (Rijndael) and others (Serpent, MARS, Twofish, RC6)
  - synergy between symbolic execution and proof
  - Slind and students (Utah)
- Memory models
  - general model applied to Java threads
  - Slind/Gopalakrishnan and students (Utah)
- ARM processor verification
  - programmers view of ARM6 equivalent to pipelined microarchitecture
  - Fox (Cambridge), Birtwistle and students (Leeds) and ARM
  - future work is ESL verification using ARM model
- Verification of probabilistic algorithms
  - Miller-Rabin probabilistic primality test
  - Hurd (Cambridge)
- Mechanised semantics of realistic networking (UDP)
  - validate operational semantics of network programming protocols
  - Sewell/Wansbrough & Norrish (Cambridge & Australia)