Automated Reasoning and Formal Verification

- ?
- ?
- ?
- ?
- ?
- ?

Automated Reasoning and Formal Verification

- What is verification by automated reasoning?
- Direct theorem proving versus embedded theorem proving
- Examples (Fox, Hurd, Slind)
- Theorem provers as tool implementation platforms
- Debugging versus proof of correctness, proof as IP
- Conclusions, opinions

What is verification by automated reasoning

- Use of a theorem prover to aid verification.
  Here's an arbitrary selection of applications:

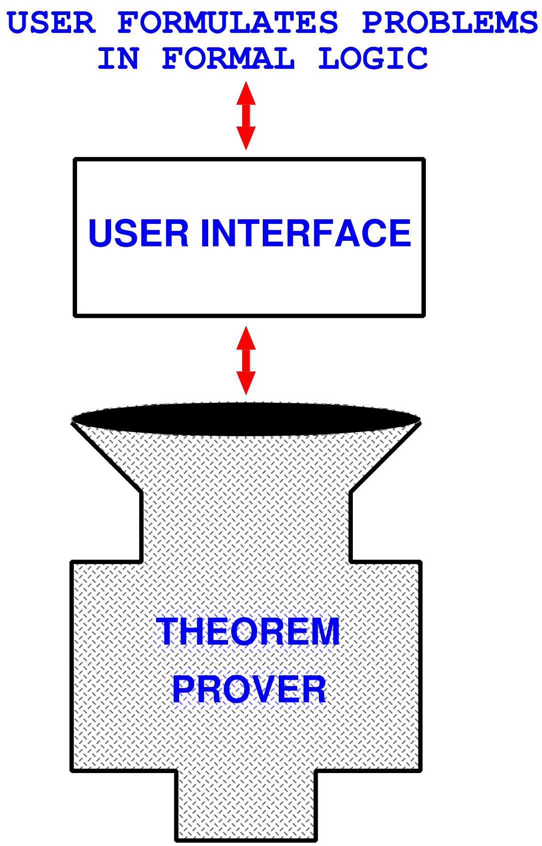
  parts of processors (e.g. pipelines, floating point units), whole processors, crypto hardware, security protocols, synchronization protocols, distributed algorithms, synthesis, system properties (e.g. separation), compilers, code transformation, high level code, machine code, proof carrying code, meta-theorems about property/hardware/software/design languages, flight control systems, railway signalling, ...

- Broad interpretation of theorem proving includes most FV methods

Verification task	Theorem proving technique	Theorems proved
boolean equivalence	propositional algorithms (BDD, SAT etc)	$\vdash(B_1 = B_2)$
model checking	fixpoint calculation, automata algorithms etc	$\vdash(\mathcal{M} \models P)$
assertion checking	decision procedures, first-order methods	$\vdash f$
proof of correctness	induction, heuristic search, interactive proof	$\vdash \mathcal{F}$

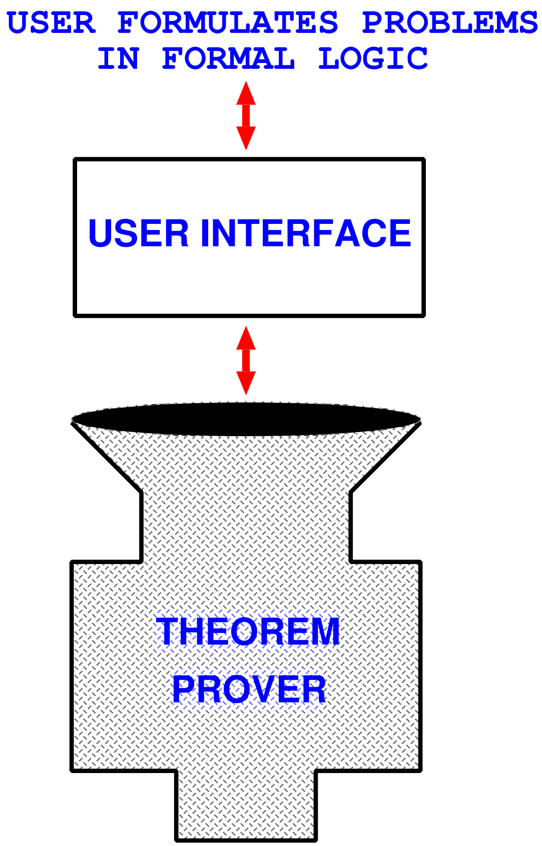
Direct versus embedded theorem proving

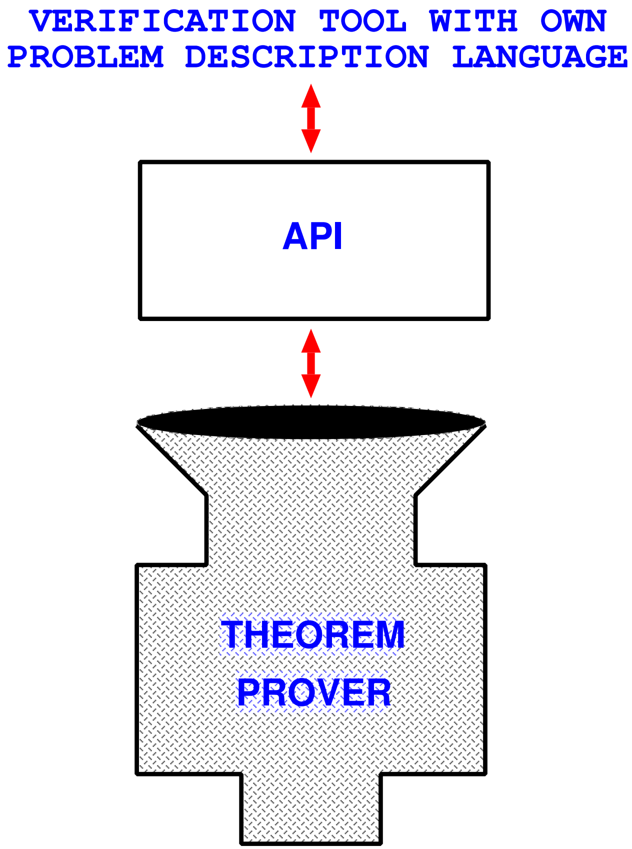
- Theorem prover can be used directly



Direct versus embedded theorem proving

- Theorem prover can be used directly . . . . or embedded in a tool





Direct and embedded theorem proving

- Direct proving mainly for traditional heroic proofs

- Embedded proving common for cool new verification applications

Direct and embedded theorem proving

- Direct proving mainly for traditional heroic proofs
  - substantial user guidance needed
  - e.g. processor proofs, verification of floating point algorithms
  - e.g. non verification proofs: Gödel's theorem, consistency of AC

- Embedded proving common for cool new verification applications

Direct and embedded theorem proving

- Direct proving mainly for traditional heroic proofs
  - substantial user guidance needed
  - e.g. processor proofs, verification of floating point algorithms
  - e.g. non verification proofs: Gödel's theorem, consistency of AC

- Embedded proving common for cool new verification applications
  - can invoke automatic 'proof engines'
  - hides formal logic stuff
  - slot into standard design/verification flows

Direct and embedded theorem proving

- Direct proving mainly for traditional heroic proofs
  - substantial user guidance needed
  - e.g. processor proofs, verification of floating point algorithms
  - e.g. non verification proofs: Gödel's theorem, consistency of AC
  - Example: ARM6 verification

- Embedded proving common for cool new verification applications
  - can invoke automatic 'proof engines'
  - hides formal logic stuff
  - slot into standard design/verification flows
  - Example: PSL/Sugar semantics directed tools

Proving processors correct

- Traditional application of theorem proving since 1980s

Proving processors correct

- Traditional application of theorem proving since 1980s
  - academic processors:
    Tamarack, Viper, SECD, FM8501/FM9001, VAMP
  - fragments of commercial processors:
    Sun, HP, Intel, IBM, AMD, Rockwell-Collins

Proving processors correct

- Traditional application of theorem proving since 1980s
  - academic processors:
    Tamarack, Viper, SECD, FM8501/FM9001, VAMP
  - fragments of commercial processors:
    Sun, HP, Intel, IBM, AMD, Rockwell-Collins
- Now feasible for complete commercial off-the-shelf (COTS) processors

Proving processors correct

- Traditional application of theorem proving since 1980s
  - academic processors:
    Tamarack, Viper, SECD, FM8501/FM9001, VAMP
  - fragments of commercial processors:
    Sun, HP, Intel, IBM, AMD, Rockwell-Collins
- Now feasible for complete commercial off-the-shelf (COTS) processors
  - improved theorem proving
  - small processors for mobile applications

Proving processors correct

- Traditional application of theorem proving since 1980s
  - academic processors:
    Tamarack, Viper, SECD, FM8501/FM9001, VAMP
  - fragments of commercial processors:
    Sun, HP, Intel, IBM, AMD, Rockwell-Collins
- Now feasible for complete commercial off-the-shelf (COTS) processors
  - improved theorem proving
  - small processors for mobile applications
- But is it worthwhile?

Example: specification and verification of ARM6 (Anthony Fox, Cambridge)

- Implementatations of all instructions of ARM6 formally verified
- What have we learned?
- What does ARM think?

Example: specification and verification of ARM6 (Anthony Fox, Cambridge)

- Implementatations of all instructions of ARM6 formally verified
  - abstractions of a cycle-accurate pipeline implementations
  - took about a person year . . . . fairly traditional heroic proof

- What have we learned?

- What does ARM think?

Example: specification and verification of ARM6 (Anthony Fox, Cambridge)

- Implementatations of all instructions of ARM6 formally verified
  - abstractions of a cycle-accurate pipeline implementations
  - took about a person year . . . . fairly traditional heroic proof
    - used the “Swansea algebraic method” of Harman & Tucker
    - could have been done 15 years ago (with much more effort)
- What have we learned?
- What does ARM think?

Example: specification and verification of ARM6 (Anthony Fox, Cambridge)

- Implementatations of all instructions of ARM6 formally verified
  - abstractions of a cycle-accurate pipeline implementations
  - took about a person year . . . . fairly traditional heroic proof
    - used the “Swansea algebraic method” of Harman & Tucker
    - could have been done 15 years ago (with much more effort)
- What have we learned?
  - time needed to prove correct small COTS processors
  - value of effective symbolic execution (already known by Boyer/Moore)
  - lots of low level ‘how to’ details
- What does ARM think?

Example: specification and verification of ARM6 (Anthony Fox, Cambridge)

- Implementatations of all instructions of ARM6 formally verified
  - abstractions of a cycle-accurate pipeline implementations
  - took about a person year . . . . fairly traditional heroic proof
    - used the “Swansea algebraic method” of Harman & Tucker
    - could have been done 15 years ago (with much more effort)
- What have we learned?
  - time needed to prove correct small COTS processors
  - value of effective symbolic execution (already known by Boyer/Moore)
  - lots of low level ‘how to’ details
- What does ARM think?
  - unimpressed by time taken
  - verification is debugging, not assurance

Debugging versus assurance: opinions are divided

- Find bugs, not proofs

- Real value is assurance that there are no bugs

Debugging versus assurance: opinions are divided

- Find bugs, not proofs

  Proofs have low value. Counter-examples have very high value.
  Counter-example technologies have seen tremendous advances over last few years.
  Proof technologies have not made much progress.
  Design teams that try a revolutionary path (e.g., “proving correctness”) will miss their next tapeouts and be out of business (or out of jobs).
  [http://www.0-in.com/papers/DAC02Pr.PDF]

- Real value is assurance that there are no bugs

Debugging versus assurance: opinions are divided

- Find bugs, not proofs

  Proofs have low value. Counter-examples have very high value.
  Counter-example technologies have seen tremendous advances over last few years.
  Proof technologies have not made much progress. .... false!
  Design teams that try a revolutionary path (e.g., “proving correctness”) will miss their next tapeouts and be out of business (or out of jobs).
  [http://www.0-in.com/papers/DAC02Pr.PDF]

- Real value is assurance that there are no bugs

Debugging versus assurance: opinions are divided

- Find bugs, not proofs

  Proofs have low value. Counter-examples have very high value.
  Counter-example technologies have seen tremendous advances over last few years.
  Proof technologies have not made much progress. .... false!
  Design teams that try a revolutionary path (e.g., "proving correctness") will miss their next tapeouts and be out of business (or out of jobs).
  [http://www.0-in.com/papers/DAC02Pr.PDF]

- Real value is assurance that there are no bugs

  ... senior staff engineer at XXXX, said formal verification has two possible applications finding bugs in RTL code, and gaining assurance of zero bugs prior to tapeout. "What we've found at XXXX, although we do find bugs, is that the real value of formal verification is the assurance," ...
  [http://www.eedesign.com/story/OEG20030606S0017]

My opinions

- Finding bugs has immediate value

My opinions

- Finding bugs has immediate value

  Rational people resist change as long as they can get the job done using current methods.
  [http://www.0-in.com/papers/DAC02Pr.PDF]

My opinions

- Finding bugs has immediate value

  Rational people resist change as long as they can get the job done using current methods.
  [http://www.0-in.com/papers/DAC02Pr.PDF]

- Proof can deliver more

My opinions

- Finding bugs has immediate value

  Rational people resist change as long as they can get the job done using current methods.
  [http://www.0-in.com/papers/DAC02Pr.PDF]

- Proof can deliver more
  - 100% coverage
  - added value, especially in security applications

My opinions

- Finding bugs has immediate value

  Rational people resist change as long as they can get the job done using current methods.
  [http://www.0-in.com/papers/DAC02Pr.PDF]

- Proof can deliver more
  - 100% coverage
  - added value, especially in security applications

- Full correctness assurance is possible now, and the cost is falling!

My opinions

- Finding bugs has immediate value

  Rational people resist change as long as they can get the job done using current methods.
  [http://www.0-in.com/papers/DAC02Pr.PDF]

- Proof can deliver more
  - 100% coverage
  - added value, especially in security applications

- Full correctness assurance is possible now, and the cost is falling!
  - theorem proving methods getting better and better
  - computers faster and cheaper, so deep proof search more practical
  - reusable IP needs specifications with correctness assurance

From debugging to assurance

- Current debugging flow (functional)

From debugging to assurance

- Current debugging flow (functional)
  - first find obvious bugs (initial sanity checking)
  - next do systematic debugging using coverage tools (Vera, Specman)
  - use FV point tools to find ‘high value’ bugs

From debugging to assurance

- Current debugging flow (functional)
  - first find obvious bugs (initial sanity checking)
  - next do systematic debugging using coverage tools (Vera, Specman)
  - use FV point tools to find ‘high value’ bugs
- Incrementally add

From debugging to assurance

- Current debugging flow (functional)
  - first find obvious bugs (initial sanity checking)
  - next do systematic debugging using coverage tools (Vera, Specman)
  - use FV point tools to find ‘high value’ bugs
- Incrementally add
  - correctness of high level algorithms (e.g. floating point)
  - bigger properties by deductively combining results of traditional FV

From debugging to assurance

- Current debugging flow (functional)
  - first find obvious bugs (initial sanity checking)
  - next do systematic debugging using coverage tools (Vera, Specman)
  - use FV point tools to find ‘high value’ bugs
- Incrementally add
  - correctness of high level algorithms (e.g. floating point)
  - bigger properties by deductively combining results of traditional FV
- Need smooth transition from debugging to assurance

From debugging to assurance

- Current debugging flow (functional)
  - first find obvious bugs (initial sanity checking)
  - next do systematic debugging using coverage tools (Vera, Specman)
  - use FV point tools to find 'high value' bugs
- Incrementally add
  - correctness of high level algorithms (e.g. floating point)
  - bigger properties by deductively combining results of traditional FV
- Need smooth transition from debugging to assurance
  - current testbench tools have some FV (property checking)
  - next generation testbenches should have access to theorem proving
  - move from coverage metrics to total coverage

From debugging to assurance

- Current debugging flow (functional)
  - first find obvious bugs (initial sanity checking)
  - next do systematic debugging using coverage tools (Vera, Specman)
  - use FV point tools to find 'high value' bugs
- Incrementally add
  - correctness of high level algorithms (e.g. floating point)
  - bigger properties by deductively combining results of traditional FV
- Need smooth transition from debugging to assurance
  - current testbench tools have some FV (property checking)
  - next generation testbenches should have access to theorem proving
  - move from coverage metrics to total coverage
- How to combine debugging FV with assurance theorem proving?

Current research on theorem proving for FV

- Add checking and simulation to a theorem prover
- Add theorem proving to a model checker
- Build new verification platform

Current research on theorem proving for FV

- Add checking and simulation to a theorem prover
  - start with user guided prover, add fast execution & model checking
  - efficiently decide properties in subset of a powerful logic
  - Examples: Acl2, PVS, HOL
- Add theorem proving to a model checker
- Build new verification platform

Current research on theorem proving for FV

- Add checking and simulation to a theorem prover
  - start with user guided prover, add fast execution & model checking
  - efficiently decide properties in subset of a powerful logic
  - Examples: Acl2, PVS, HOL
- Add theorem proving to a model checker
  - start with property checker, add deductive layer
  - combine properties, manage abstraction
  - Examples: Cadence SMV, Intel's ThmTac
- Build new verification platform

Current research on theorem proving for FV

- Add checking and simulation to a theorem prover
  - start with user guided prover, add fast execution & model checking
  - efficiently decide properties in subset of a powerful logic
  - Examples: Acl2, PVS, HOL
- Add theorem proving to a model checker
  - start with property checker, add deductive layer
  - combine properties, manage abstraction
  - Examples: Cadence SMV, Intel's ThmTac
- Build new verification platform
  - Example: Intel's Forte/$reFL^{ect}$

Current research on theorem proving for FV

- Add checking and simulation to a theorem prover
  - start with user guided prover, add fast execution & model checking
  - efficiently decide properties in subset of a powerful logic
  - Examples: Acl2, PVS, HOL
- Add theorem proving to a model checker
  - start with property checker, add deductive layer
  - combine properties, manage abstraction
  - Examples: Cadence SMV, Intel's ThmTac
- Build new verification platform
  - Example: Intel's Forte/$reFL^{ect}$
- Issues

Current research on theorem proving for FV

- Add checking and simulation to a theorem prover
  - start with user guided prover, add fast execution & model checking
  - efficiently decide properties in subset of a powerful logic
  - Examples: Acl2, PVS, HOL
- Add theorem proving to a model checker
  - start with property checker, add deductive layer
  - combine properties, manage abstraction
  - Examples: Cadence SMV, Intel's ThmTac
- Build new verification platform
  - Example: Intel's Forte/$reFL^{ect}$
- Issues
  - security versus efficiency (assurance versus debugging)
  - programmability (ease-of-use versus fliexibility and power)

Example: HOL4

- Start with interactive ‘LCF style’ proof assistant for higher order logic

Example: HOL4

- Start with interactive ‘LCF style’ proof assistant for higher order logic
  - theorem tagging to track provenance

Example: HOL4

- Start with interactive ‘LCF style’ proof assistant for higher order logic
  - theorem tagging to track provenance
- Add execution capability

Example: HOL4

- Start with interactive ‘LCF style’ proof assistant for higher order logic
  - theorem tagging to track provenance
- Add execution capability
  - modified kernel to handle explicit substitutions (Barras)
  - future: faster non fully expansive execution via compilation (Slind)

Example: HOL4

- Start with interactive 'LCF style' proof assistant for higher order logic
  - theorem tagging to track provenance
- Add execution capability
  - modified kernel to handle explicit substitutions (Barras)
  - future: faster non fully expansive execution via compilation (Slind)
- Add property checking technology

Example: HOL4

- Start with interactive ‘LCF style’ proof assistant for higher order logic
  - theorem tagging to track provenance
- Add execution capability
  - modified kernel to handle explicit substitutions (Barras)
  - future: faster non fully expansive execution via compilation (Slind)
- Add property checking technology
  - BDD representations for QBFs
  - external BDD package for specialised rules (e.g. image computation)
  - file interface to SAT packages (zChaff, Grasp, SATO)

Example: HOL4

- Start with interactive 'LCF style' proof assistant for higher order logic
  - theorem tagging to track provenance
- Add execution capability
  - modified kernel to handle explicit substitutions (Barras)
  - future: faster non fully expansive execution via compilation (Slind)
- Add property checking technology
  - BDD representations for QBFs
  - external BDD package for specialised rules (e.g. image computation)
  - file interface to SAT packages (zChaff, Grasp, SATO)
- Applications

Example: HOL4

- Start with interactive ‘LCF style’ proof assistant for higher order logic
  - theorem tagging to track provenance
- Add execution capability
  - modified kernel to handle explicit substitutions (Barras)
  - future: faster non fully expansive execution via compilation (Slind)
- Add property checking technology
  - BDD representations for QBFs
  - external BDD package for specialised rules (e.g. image computation)
  - file interface to SAT packages (zChaff, Grasp, SATO)
- Applications
  - fully expansive model checker (BDDs + SAT for refinements) – Amjad
  - `puzzleTool`: rewrite puzzle descriptions to QBFs, solve with BDDs

Example: executing the formal semantics of PSL/Sugar

(joint work with Joe Hurd & Konrad Slind)

- Show formal semantics is not just documentation

Example: executing the formal semantics of PSL/Sugar

(joint work with Joe Hurd & Konrad Slind)

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

Example: executing the formal semantics of PSL/Sugar

(joint work with Joe Hurd & Konrad Slind)

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

- Correctness primary, efficiency secondary

Example: executing the formal semantics of PSL/Sugar

(joint work with Joe Hurd & Konrad Slind)

- Show formal semantics is not just documentation
  - can run the Language Reference Manual (LRM)
- Correctness primary, efficiency secondary
  - but need sufficient efficiency!

Example: executing the formal semantics of PSL/Sugar

(joint work with Joe Hurd & Konrad Slind)

- 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

Example: executing the formal semantics of PSL/Sugar

(joint work with Joe Hurd & Konrad Slind)

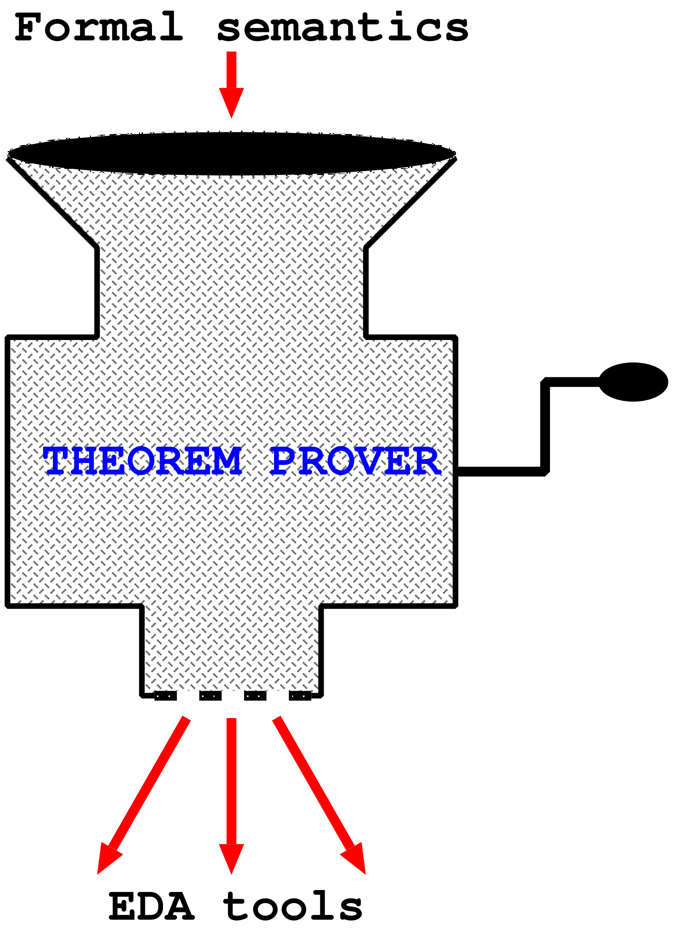
- Show formal semantics is not just documentation
  - can run the Language Reference Manual (LRM)

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

- Programming methodology, not new verification algorithms
  - EDA tools with theorem prover inside (*c.f.* PROSPER)

Use theorem proving to generate tools from semantics

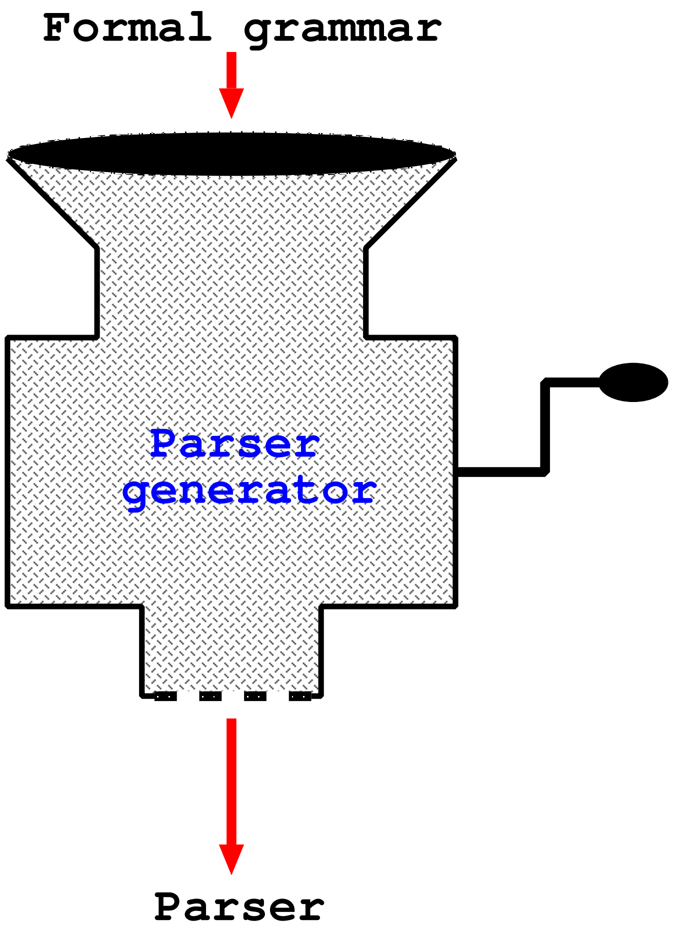
Use theorem proving to generate tools from semantics



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

Compare with generating tools from syntax

Compare with generating tools from syntax

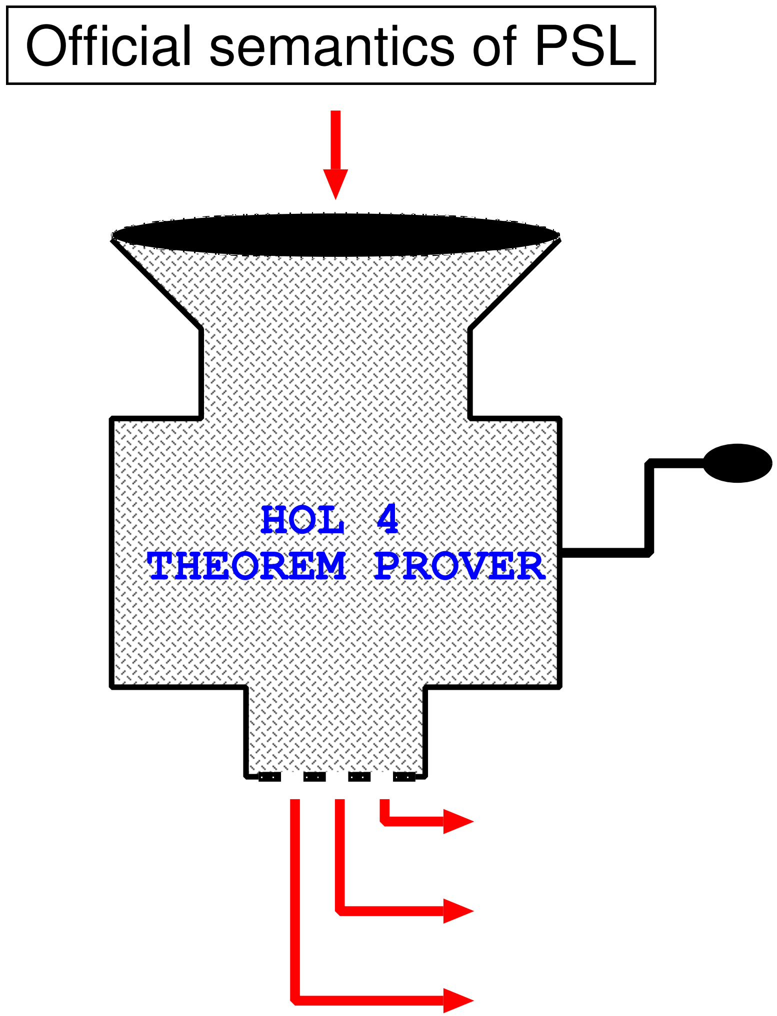


- Input a grammar
- Apply theory of formal languages
- Generate a parser

Accellera's PSL (formerly IBM's Sugar 2.0)

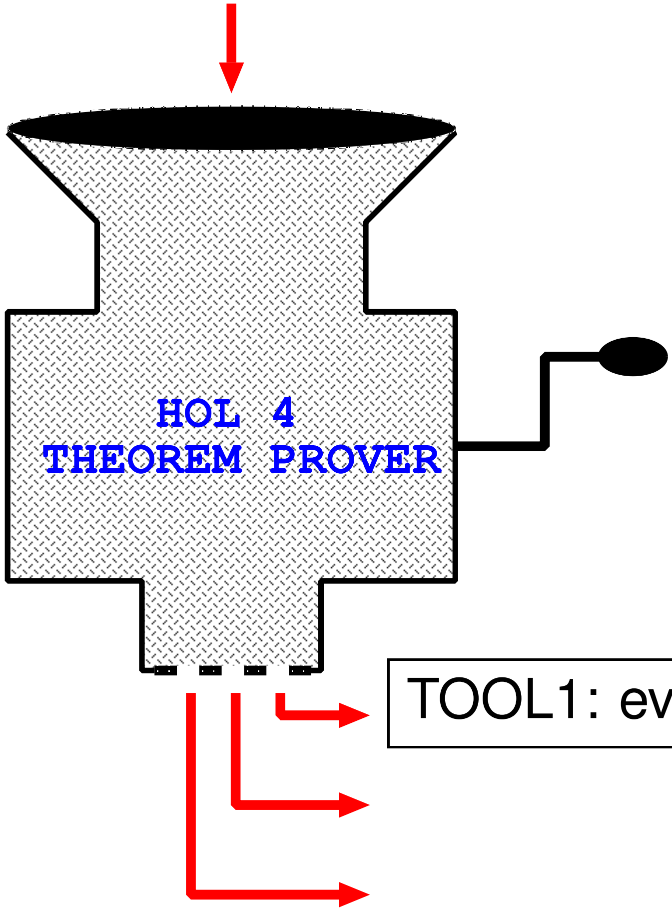
- PSL is a property specification language combining
  - boolean expressions (Verilog syntax)
  - patterns (Sequential Extended Regular Expressions SEREs)
  - LTL formulas (Foundation language FL)
  - CTL formulas (Optional Branching Extension OBE)
- Designed both for model checking and simulation testbenches
- Intended to be the industry standard

Generating PSL tools



Generating PSL tools

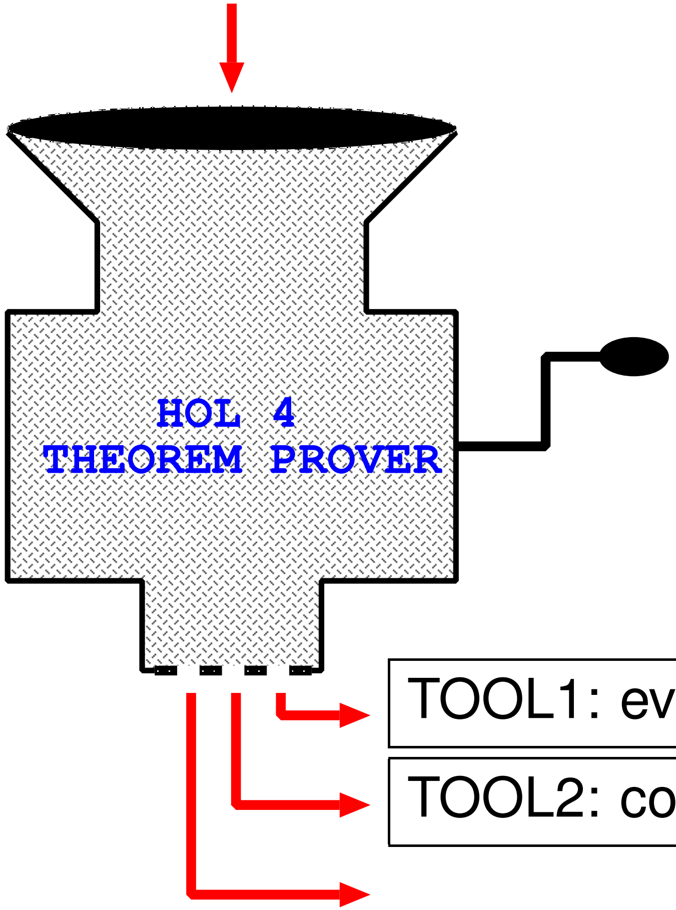
Official semantics of PSL



TOOL1: evaluate properties on a specific path

Generating PSL tools

Official semantics of PSL

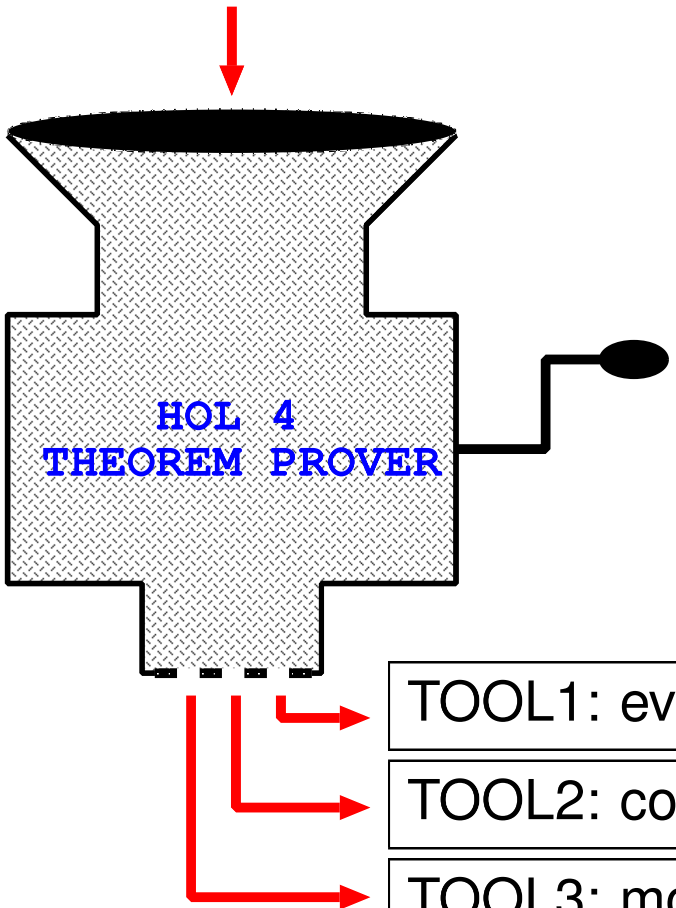


TOOL1: evaluate properties on a specific path

TOOL2: compile properties to HDL checkers (Hurd)

Generating PSL tools

Official semantics of PSL



TOOL1: evaluate properties on a specific path

TOOL2: compile properties to HDL checkers (Hurd)

TOOL3: model check OBE properties (Amjad's PhD)

TOOL1: executing the semantics

- By rewriting and evaluation (PSL in red, HOL in blue):

  $\vdash$

  $\vdash$

TOOL1: executing the semantics

- By rewriting and evaluation (PSL in red, HOL in blue):

$\vdash\ s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 \models p \wedge \texttt{next!} f \quad = \quad s_0 \models p\ \wedge\ s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 \models f$

$\vdash$

TOOL1: executing the semantics

- By rewriting and evaluation (PSL in red, HOL in blue):

  $\vdash\ s_0s_1s_2s_3s_4s_5s_6s_7s_8s_9 \models p \wedge \texttt{next!}f \ = \ s_0 \models p \ \wedge \ s_1s_2s_3s_4s_5s_6s_7s_8s_9 \models f$

  $\vdash\ \{a\}\{a,b\}\{b\} \models a \wedge \texttt{next!}b \ = \ \mathsf{T}$

TOOL1: executing the semantics

- By rewriting and evaluation (PSL in red, HOL in blue):

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

  $\vdash\ \{a\}\{a,b\}\{b\} \models a \wedge \mathtt{next!}\, b \ = \ \mathsf{T}$

- LRM semantics of the until-operator not directly executable

TOOL1: executing the semantics

- By rewriting and evaluation (PSL in red, HOL in blue):

$\vdash\ s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 \models p \wedge \texttt{next!} f \ = \ s_0 \models p \ \wedge \ s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 \models f$

$\vdash\ \{a\}\{a,b\}\{b\} \models a \wedge \texttt{next!} b \ = \ \mathsf{T}$

- LRM semantics of the until-operator not directly executable

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

TOOL1: executing the semantics

- By rewriting and evaluation (PSL in red, HOL in blue):

  $\vdash\ s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 \models p \wedge \texttt{next!} f \ \ = \ \ s_0 \models p \ \wedge \ s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 \models f$

  $\vdash\ \{a\}\{a,b\}\{b\} \models a \wedge \texttt{next!} b \ = \ \mathsf{T}$

- LRM semantics of the until-operator not directly executable

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

- Standard reformulation makes it directly executable

TOOL1: executing the semantics

- By rewriting and evaluation (PSL in red, HOL in blue):

  $\vdash\ s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 \models p \wedge \texttt{next!} f \ = \ s_0 \models p \ \wedge \ s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 \models f$

  $\vdash\ \{a\}\{a,b\}\{b\} \models a \wedge \texttt{next!} b \ = \ \mathsf{T}$

- 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

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

TOOL1: executing the semantics

- By rewriting and evaluation (PSL in red, HOL in blue):

  $\vdash\ s_0s_1s_2s_3s_4s_5s_6s_7s_8s_9 \models p \land \texttt{next!}f \ = \ s_0 \models p \ \land \ s_1s_2s_3s_4s_5s_6s_7s_8s_9 \models f$

  $\vdash\ \{a\}\{a,b\}\{b\} \models a \land \texttt{next!}b \ = \ \mathsf{T}$

- 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:

TOOL1: executing the semantics

- By rewriting and evaluation (PSL in red, HOL in blue):

  $\vdash\ s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 \models p \wedge \texttt{next!}f \ = \ s_0 \models p \ \wedge \ s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 \models f$

  $\vdash\ \{a\}\{a,b\}\{b\} \models a \wedge \texttt{next!}b \ = \ \mathsf{T}$

- LRM semantics of the until-operator not directly executable

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

- Standard reformulation makes it directly executable

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

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

  $\vdash\ s_0 s_1 s_2 s_3 s_4 \models [b_1\ U\ b_2] =$

  $s_0 \models b_2 \vee$

  $s_0 \models b_1 \wedge (s_1 \models b_2 \vee s_1 \models b_1 \wedge$

  $(s_2 \models b_2 \vee s_2 \models b_1 \wedge (s_3 \models b_2 \vee s_3 \models b_1 \wedge s_4 \models b_2)))$

Matching regular expressions by proof (Hurd)

- PSL formulas may contain regular expressions

Matching regular expressions by proof (Hurd)

- PSL formulas may contain regular expressions
- Semantics of PSL regular expressions is self-explanatory

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

Matching regular expressions by proof (Hurd)

- PSL formulas may contain regular expressions
- Semantics of PSL regular expressions is self-explanatory

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

- Make executable by proving

Matching regular expressions by proof (Hurd)

- PSL formulas may contain regular expressions
- Semantics of PSL regular expressions 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

  $\vdash\ \ \forall\, w\ r.\ w \models r\ \ =\ \ \mathsf{Match}\ r\ w$

  where:

  - Match is an executable matcher for regular expressions

Example formula with regular expression: $\{r\}(f)$ (Hurd)

- Called "suffix implication", semantics is:

  $w \models \{r\}(f) \;=\; \forall j \in [0 \mathinner{..} |w|).\ w^{0,j} \models r \Rightarrow w^j \models f$

Example formula with regular expression: $\{r\}(f)$ (Hurd)

- Called "suffix implication", semantics is:

  $w \models \{r\}(f) \ = \ \forall j \in [0\,..\,|w|).\ w^{0,j} \models r \Rightarrow w^j \models f$

- Define an efficient executable function Check so that, for example:

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

Example formula with regular expression: $\{r\}(f)$ (Hurd)

- Called “suffix implication”, semantics is:

  $w \models \{r\}(f) \;=\; \forall j \in [0\,..\,|w|).\; w^{0,j} \models r \Rightarrow w^j \models f$

- Define an efficient executable function Check so that, for example:

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

- Then prove

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

Example formula with regular expression: $\{r\}(f)$ (Hurd)

- Called "suffix implication", semantics is:

  $$w \models \{r\}(f) \ = \ \forall j \in [0 \mathinner{..} |w|).\ w^{0,j} \models r \Rightarrow w^j \models f$$

- Define an efficient executable function Check so that, for example:

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

- Then prove

  $$\vdash \ \forall w\ r\ f.\ w \models \{r\}(f) \ = \ \mathsf{Check}\ r\ (\lambda x.\ x \models f)\ w$$

- Rewrite with this, then execute

Example illustrating TOOL1

- 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}`

- Can evaluate in SML, or via a command line wrapper

- Example: to evaluate `(c && 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 $w \models^{\mathsf{T}} f$ from semantics)

Uses of TOOL1 (calculating $w \models^{\mathsf{T}} f$ from semantics)

- Teaching and learning tool for exploring semantics

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

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)

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

Tools use standard algorithms

- TOOL1: semantic calculator
- TOOL2: checker compiler (Hurd)
- TOOL3: symbolic μ-calculus model checker (Amjad)

Tools use standard algorithms

- TOOL1: semantic calculator
  - match regexps using automata; evaluate formulas recursively
- TOOL2: checker compiler (Hurd)
- TOOL3: symbolic μ-calculus model checker (Amjad)

Tools use standard algorithms

- TOOL1: semantic calculator
  - match regexps using automata; evaluate formulas recursively
- TOOL2: checker compiler (Hurd)
  - compile regexps to automata, then ‘pretty print’ to HDL (Verilog)
- TOOL3: symbolic μ-calculus model checker (Amjad)

Tools use standard algorithms

- TOOL1: semantic calculator
  - match regexps using automata; evaluate formulas recursively
- TOOL2: checker compiler (Hurd)
  - compile regexps to automata, then ‘pretty print’ to HDL (Verilog)
- TOOL3: symbolic μ-calculus model checker (Amjad)
  - use BDD representation judgements to link HOL terms to BDDs

Tools use standard algorithms

- TOOL1: semantic calculator
  - match regexps using automata; evaluate formulas recursively
- TOOL2: checker compiler (Hurd)
  - compile regexps to automata, then ‘pretty print’ to HDL (Verilog)
- TOOL3: symbolic μ-calculus model checker (Amjad)
  - use BDD representation judgements to link HOL terms to BDDs
- No new algorithms, but maybe a new kind of logic programming

Summary

Summary

- Verification tools use theorem proving
  - code derived (manually) from semantics by proof
  - tools use embedded theorem proving when they execute

Summary

- Verification tools use theorem proving
  - code derived (manually) from semantics by proof
  - tools use embedded theorem proving when they execute
- Correct by construction
  - PSL semantics will evolve for at least another year

Summary

- Verification tools use theorem proving
  - code derived (manually) from semantics by proof
  - tools use embedded theorem proving when they execute
- Correct by construction
  - PSL semantics will evolve for at least another year
- Theorem prover as implementation platform

Summary

- Verification tools use theorem proving
  - code derived (manually) from semantics by proof
  - tools use embedded theorem proving when they execute
- Correct by construction
  - PSL semantics will evolve for at least another year
- Theorem prover as implementation platform
  - prototyping standards compliant tools
  - theorem proving is slow
  - maybe OK for some industrial strength *performance-non-critical* tools

Summary

- Verification tools use theorem proving
  - code derived (manually) from semantics by proof
  - tools use embedded theorem proving when they execute
- Correct by construction
  - PSL semantics will evolve for at least another year
- Theorem prover as implementation platform
  - prototyping standards compliant tools
  - theorem proving is slow .... but not necessarily too slow
  - maybe OK for some industrial strength *performance-non-critical* tools

Future research idea: generating ESL design tools

- SANTA CLARA, Calif., June 16, 2003 XXXX, Inc. the leading supplier of configurable and extensible microprocessor cores, today introduced XXXX XXXX, the first integrated design environment (IDE) for SOC development that integrates software development, processor optimization and multiple-processor system-on-chip (SOC) architecture tools into one common design environment. XXXXs XXXX IDE is a visual environment with a host of new automation tools that makes creating XXX processor-based SOC hardware and software much easier. [http://www.tensilica.com/html/pr_2003_06_16a.html]

Future research idea: generating ESL design tools

- SANTA CLARA, Calif., June 16, 2003 XXXX, Inc. the leading supplier of configurable and extensible microprocessor cores, today introduced XXXX XXXX, the first integrated design environment (IDE) for SOC development that integrates software development, processor optimization and multiple-processor system-on-chip (SOC) architecture tools into one common design environment. XXXXs XXXX IDE is a visual environment with a host of new automation tools that makes creating XXX processor-based SOC hardware and software much easier.
  [http://www.tensilica.com/html/pr_2003_06_16a.html]

- Generate bespoke verifier for XXX processor?
  - input processor specification
  - generate analysis tools for XXX-based ESL platform
  - not performace critical, so implementation by deduction plausible

Could there be a market for 'Proof IP'

- Already exists design IP and property IP

Could there be a market for ‘Proof IP’

- Already exists design IP and property IP
  - e.g. ARM designs and AMBA golden properties

Could there be a market for ‘Proof IP’

- Already exists design IP and property IP
  - e.g. ARM designs and AMBA golden properties
- What about high level specification and proof IP?

Could there be a market for ‘Proof IP’

- Already exists design IP and property IP
  - e.g. ARM designs and AMBA golden properties
- What about high level specification and proof IP?
  - design IP needs multilevel specifications (RTL, behavioral)

Could there be a market for ‘Proof IP’

- Already exists design IP and property IP
  - e.g. ARM designs and AMBA golden properties
- What about high level specification and proof IP?
  - design IP needs multilevel specifications (RTL, behavioral)
- Specifications are more valuable if correct

Could there be a market for 'Proof IP'

- Already exists design IP and property IP
  - e.g. ARM designs and AMBA golden properties
- What about high level specification and proof IP?
  - design IP needs multilevel specifications (RTL, behavioral)
- Specifications are more valuable if correct
  - sell validated component specifications

Could there be a market for 'Proof IP'

- Already exists design IP and property IP
  - e.g. ARM designs and AMBA golden properties
- What about high level specification and proof IP?
  - design IP needs multilevel specifications (RTL, behavioral)
- Specifications are more valuable if correct
  - sell validated component specifications
- Design tweaks need verification tweeks

Could there be a market for 'Proof IP'

- Already exists design IP and property IP
  - e.g. ARM designs and AMBA golden properties
- What about high level specification and proof IP?
  - design IP needs multilevel specifications (RTL, behavioral)
- Specifications are more valuable if correct
  - sell validated component specifications
- Design tweaks need verification tweeks
  - sell bespoke proof scripts to validate tweaks

Quote from the web – Proof IP?

PRODUCT OVERVIEW

XXXX: Conquers Toughest Verification Challenges with 100% Formal Proof

⋮

XXXX Pre-Built Proof Kits are available for a long list of industry standard interfaces. Pre-Built Proof Kits contain all the necessary spec-level requirements to prove interface compliance, delivering immediate benefits to users.

⋮

Future challenges

Future challenges

- Getting theorem proving into real verification flows
- Continue to advance state-of-the-art of theorem proving
- More demonstrator projects
- Make a market for specification and proof IP

Future challenges

- Getting theorem proving into real verification flows
  - Intel, AMD can do it; tough for small companies
  - proof engine deployment platform: *ProofStudio.net*
- Continue to advance state-of-the-art of theorem proving
- More demonstrator projects
- Make a market for specification and proof IP

Future challenges

- Getting theorem proving into real verification flows
  - Intel, AMD can do it; tough for small companies
  - proof engine deployment platform: *ProofStudio.net*
- Continue to advance state-of-the-art of theorem proving
  - better integrate first order reasoning, equality & decision procedures
  - go beyond first order automation
- More demonstrator projects
- Make a market for specification and proof IP

Future challenges

- Getting theorem proving into real verification flows
  - Intel, AMD can do it; tough for small companies
  - proof engine deployment platform: *ProofStudio.net*
- Continue to advance state-of-the-art of theorem proving
  - better integrate first order reasoning, equality & decision procedures
  - go beyond first order automation
- More demonstrator projects
  - hard to motivate as ‘proof of concept’ established
  - need more cost/benefit stories
- Make a market for specification and proof IP

Future challenges .... Long Live Theorem Proving!

- Getting theorem proving into real verification flows
  - Intel, AMD can do it; tough for small companies
  - proof engine deployment platform: *ProofStudio.net*
- Continue to advance state-of-the-art of theorem proving
  - better integrate first order reasoning, equality & decision procedures
  - go beyond first order automation
- More demonstrator projects
  - hard to motivate as ‘proof of concept’ established
  - need more cost/benefit stories
- Make a market for specification and proof IP
  - need a convincing “value proposition” and “ROI” story

THE END