Verification by Theorem Proving

Issues and Challenges

Mike Gordon

University of Cambridge Computer Laboratory

William Gates Building

JJ Thomson Avenue

Cambridge CB3 0FD, U.K.

e-mail: mjcg@cl.cam.ac.uk

Verification by Theorem Proving

Issues and Challenges

When I wrote the abstract:

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

Verification by Theorem Proving

Issues and Challenges

Today's talk

- What is "verification by theorem proving"?
- Direct theorem proving versus embedded theorem proving
- Choice of logic, proof methodology, proof engines, middleware
- Debugging versus proof of correctness, proof as IP
- Theorem provers as tool implementation platforms
- Conclusions, opinions

What is verification by theorem proving

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

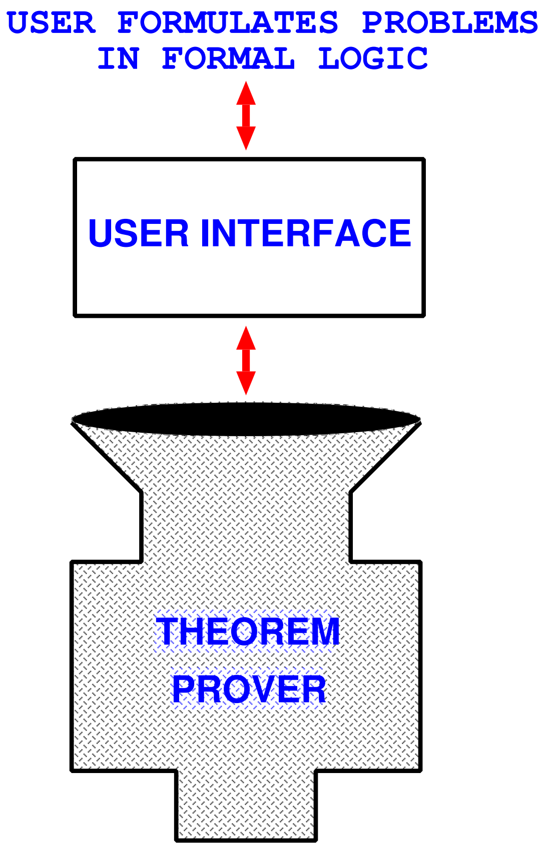
  parts of processors (e.g. pipelines, floating point units), whole processors, crypto hardware, security protocols, synchronization protocols, distributed algorithms, synthesis, system properties (e.g. separation), compilers, code transformation, high level code, machine code, proof carrying code, meta-theorems 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	$\vdash \mathcal{F}$

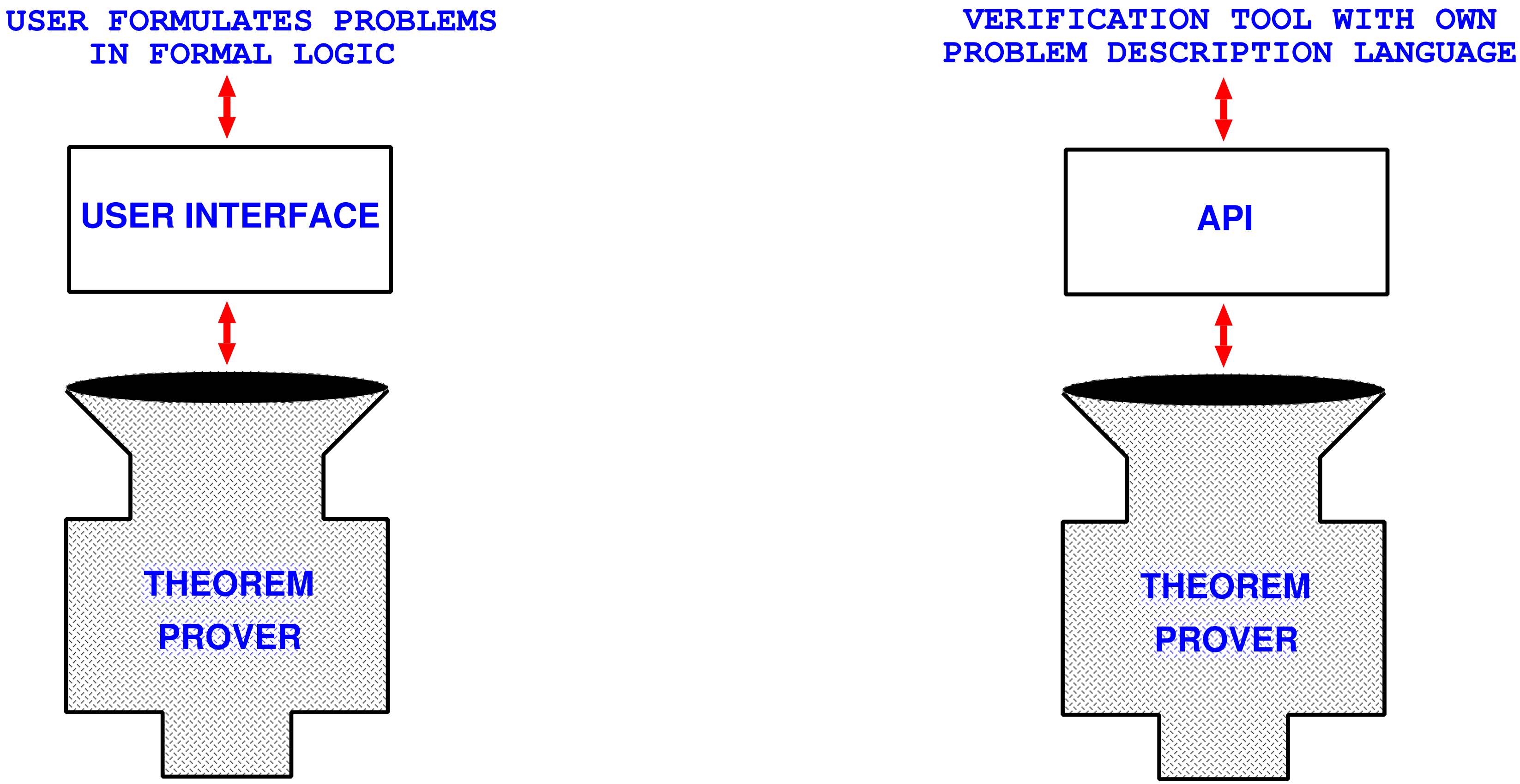
Direct versus embedded theorem proving

- 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 heroic proofs

- Embedded proving common for verification

Direct and embedded theorem proving

- Direct proving mainly for 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 verification

Direct and embedded theorem proving

- Direct proving mainly for 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 verification
  - can invoke automatic 'proof engines'
  - hides formal logic stuff
  - slot into standard design/verification flows

Direct theorem proving

- Need to formulate problems in logic
- Issue: how powerful should the logic be
- Need to drive the theorem prover to create proof
- Issue: design of user interface

Direct theorem proving

- Need to formulate problems in logic
  - can code in ‘raw logic’
  - or embed an application-specific notation
- Issue: how powerful should the logic be
- Need to drive the theorem prover to create proof
- Issue: design of user interface

Direct theorem proving

- Need to formulate problems in logic
  - can code in ‘raw logic’
  - or embed an application-specific notation
- Issue: how powerful should the logic be
  - simple logics support more automation
  - powerful logics support better specification and embedding
- Need to drive the theorem prover to create proof
- Issue: design of user interface

Direct theorem proving

- Need to formulate problems in logic
  - can code in ‘raw logic’
  - or embed an application-specific notation
- Issue: how powerful should the logic be
  - simple logics support more automation
  - powerful logics support better specification and embedding
- Need to drive the theorem prover to create proof
  - requires training, but not deep knowledge of formal logic
  - performing proof requires insight into problem
- Issue: design of user interface

Direct theorem proving

- Need to formulate problems in logic
  - can code in 'raw logic'
  - or embed an application-specific notation
- Issue: how powerful should the logic be
  - simple logics support more automation
  - powerful logics support better specification and embedding
- Need to drive the theorem prover to create proof
  - requires training, but not deep knowledge of formal logic
  - performing proof requires insight into problem
- Issue: design of user interface
  - fancy GUIs are good for beginners, but get in the way for experts
  - good for humans not necessarily good as API for tools

Recent quote from an EDA tool user group (ESNUG)

I don't know why tool vendors like GUIs so much.
They are fine for a novice user but impossible for real work.
There is just no way to script a GUI tool for regressions.
The later releases had much improved support for the command line interface.
[`http://www.deepchip.com/items/0414-05.html`]

Recent quote from an EDA tool user group (ESNUG)

I don’t know why tool vendors like GUIs so much.
They are fine for a novice user but impossible for real work.
There is just no way to script a GUI tool for regressions.
The later releases had much improved support for the command line interface.
[http://www.deepchip.com/items/0414-05.html]

- This is not about theorem proving

Recent quote from an EDA tool user group (ESNUG)

I don't know why tool vendors like GUIs so much.
They are fine for a novice user but impossible for real work.
There is just no way to script a GUI tool for regressions.
The later releases had much improved support for the command line interface.
[`http://www.deepchip.com/items/0414-05.html`]

- This is not about theorem proving ....but maybe it applies?

Choice of logic

- Specifications need both discrete and continuous mathematics

Choice of logic

- Specifications need both discrete and continuous mathematics
  - verification of floating point algorithms need real analysis
  - verification of probabilistic algorithms need measure theory

Choice of logic

- Specifications need both discrete and continuous mathematics
  - verification of floating point algorithms need real analysis
  - verification of probabilistic algorithms need measure theory
- Set theory is the standard foundation (F.O.L. + ZF axioms)

Choice of logic

- Specifications need both discrete and continuous mathematics
  - verification of floating point algorithms need real analysis
  - verification of probabilistic algorithms need measure theory
- Set theory is the standard foundation (F.O.L. + ZF axioms)
- First-order logic plus recursive datatypes seems the minimum

Choice of logic

- Specifications need both discrete and continuous mathematics
  - verification of floating point algorithms need real analysis
  - verification of probabilistic algorithms need measure theory
- Set theory is the standard foundation (F.O.L. + ZF axioms)
- First-order logic plus recursive datatypes seems the minimum
- Many provers support some kind of higher-order logic
  - because of types and support for functional programming

Choice of logic

- Specifications need both discrete and continuous mathematics
  - verification of floating point algorithms need real analysis
  - verification of probabilistic algorithms need measure theory
- Set theory is the standard foundation (F.O.L. + ZF axioms)
- First-order logic plus recursive datatypes seems the minimum
- Many provers support some kind of higher-order logic
  - because of types and support for functional programming
- Issue: should there be a standard logic?

Choice of logic

- Specifications need both discrete and continuous mathematics
  - verification of floating point algorithms need real analysis
  - verification of probabilistic algorithms need measure theory
- Set theory is the standard foundation (F.O.L. + ZF axioms)
- First-order logic plus recursive datatypes seems the minimum
- Many provers support some kind of higher-order logic
  - because of types and support for functional programming
- Issue: should there be a standard logic?
  - lots of choices:
    - ∗ first order versus higher-order
    - ∗ classical versus constructive
    - ∗ typed versus untyped

Choice of logic and embedded proving

- Switching proof methodology harder than switching language

- Maybe choice of logic not really an issue

- Embedding prover in a verification tool hides logic from users

Choice of logic and embedded proving

- Switching proof methodology harder than switching language
  - can learn a new logic in a day
  - needs a month to learn to effectively drive a major theorem prover
- Maybe choice of logic not really an issue
- Embedding prover in a verification tool hides logic from users

Choice of logic and embedded proving

- Switching proof methodology harder than switching language
  - can learn a new logic in a day
  - needs a month to learn to effectively drive a major theorem prover
- Maybe choice of logic not really an issue
  - need EDA tool builders to drive any standard
- Embedding prover in a verification tool hides logic from users

Choice of logic and embedded proving

- Switching proof methodology harder than switching language
  - can learn a new logic in a day
  - needs a month to learn to effectively drive a major theorem prover
- Maybe choice of logic not really an issue
  - need EDA tool builders to drive any standard
  - maybe next year an Accellera Theorem Proving Technical Committee!
- Embedding prover in a verification tool hides logic from users

Choice of logic and embedded proving

- Switching proof methodology harder than switching language
  - can learn a new logic in a day
  - needs a month to learn to effectively drive a major theorem prover
- Maybe choice of logic not really an issue
  - need EDA tool builders to drive any standard
  - maybe next year an Accellera Theorem Proving Technical Committee!
- Embedding prover in a verification tool hides logic from users
  - but not from tool developers
  - tool developers are usually familiar with logic and semantics

Choice of theorem prover interaction method (1)

- Orthogonal choices:
  - top-down (backward or goal-directed) versus bottom-up (forward)
  - declarative versus imperative

Choice of theorem prover interaction method (1)

- Orthogonal choices:
  - top-down (backward or goal-directed) versus bottom-up (forward)
  - declarative versus imperative
- Top-down
  - also known as “backward” or “goal-directed” proof
  - split goal-to-be-proved into subgoals
  - split subgoals into sub-subgoals
  - proceed until subgoals are decidable
- Bottom-up
  - also known as “forward proof”
  - start from axioms
  - combine using rules of inference
  - eventually deduce goal to be proved

Choice of theorem prover interaction method (2)

- Orthogonal choices:
  - top-down (backward or goal-directed) versus bottom-up (forward)
  - declarative versus imperative

Choice of theorem prover interaction method (2)

- Orthogonal choices:
  - top-down (backward or goal-directed) versus bottom-up (forward)
  - declarative versus imperative
- Declarative: give a sequence lemmas or subgoals leading to conclusion
  - for top-down, something like:
    Goal follows if *goal-1* which follows if *goal-2* · · · which follows if *goal-n* which is trivial
  - for bottom-up, something like:
    Lemma-1 hence *Lemma-2* hence · · · hence *Theorem*
  - pioneered by Mizar proof checker; like textbook proofs; readable
  - good for checking proofs, less good for finding them
- Imperative: write a proof-generating program in a 'tactic language'
  - something like:
    apply induction then simplify and invoke resolution
  - typically unreadable prover-specific instructions
  - good for finding proofs and for programming verification algorithms

Choice of theorem prover interaction method (summary)

- Top-down versus bottom-up

- Declarative versus imperative

Choice of theorem prover interaction method (summary)

- Top-down versus bottom-up
  - top-down better for direct proof
  - bottom-up good for ‘fine grain’ proof scripts (cf. machine code)
  - LCF-like systems synthesise forward proofs via backward scripts

- Declarative versus imperative

Choice of theorem prover interaction method (summary)

- Top-down versus bottom-up
  - top-down better for direct proof
  - bottom-up good for ‘fine grain’ proof scripts (cf. machine code)
  - LCF-like systems synthesise forward proofs via backward scripts
- Mature proof assistants support both directions
- Declarative versus imperative

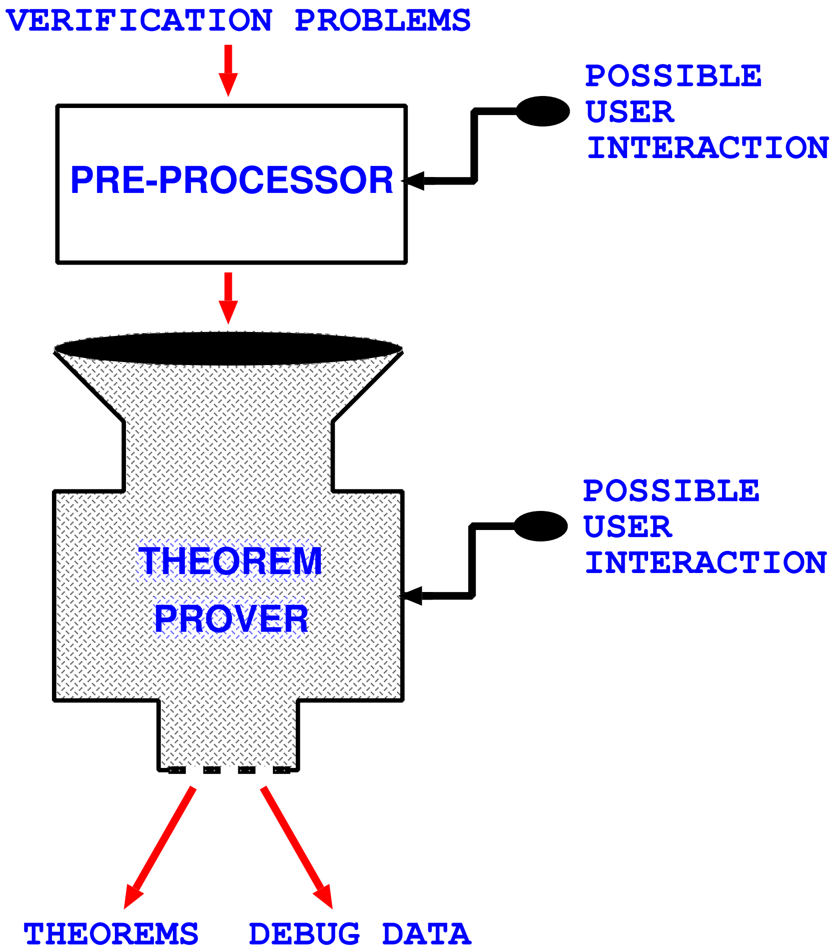
Choice of theorem prover interaction method (summary)

- Top-down versus bottom-up
  - top-down better for direct proof
  - bottom-up good for ‘fine grain’ proof scripts (cf. machine code)
  - LCF-like systems synthesise forward proofs via backward scripts
- Mature proof assistants support both directions
- Declarative versus imperative
  - imperative:
    - better for finding proofs (can invoke proof search engines)
    - provides better API for embedded proof
  - declarative:
    - more natural for checking textbook like proofs
    - but rather verbose and ‘COBOL like’

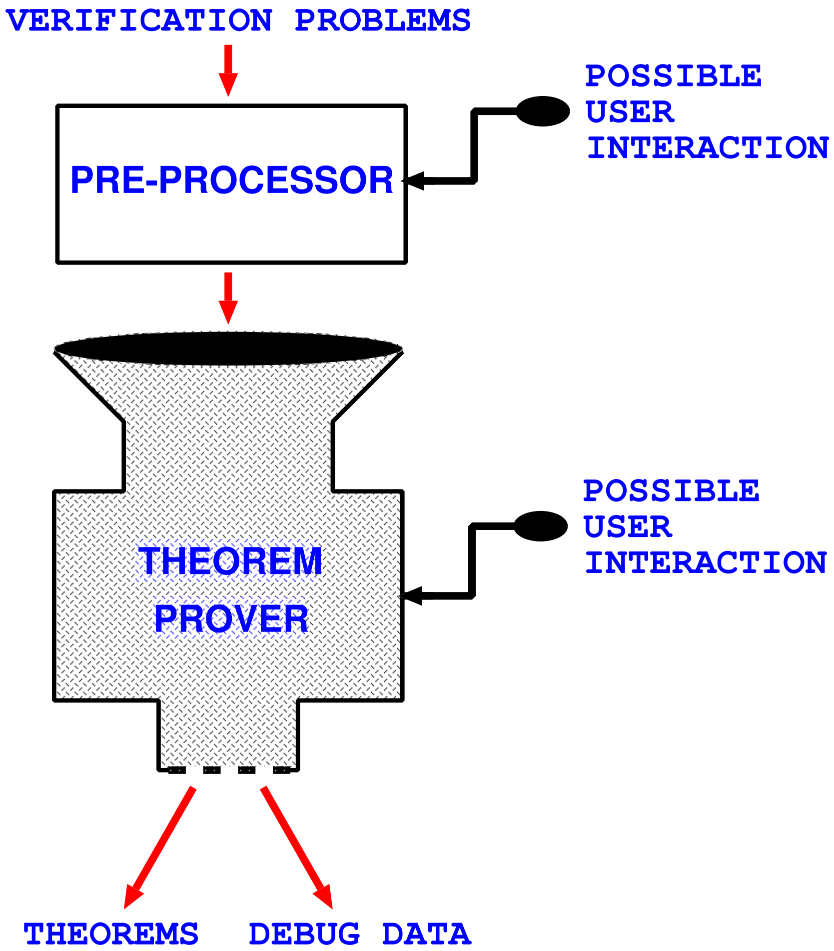
Choice of theorem prover interaction method (summary)

- Top-down versus bottom-up
  - top-down better for direct proof
  - bottom-up good for 'fine grain' proof scripts (cf. machine code)
  - LCF-like systems synthesise forward proofs via backward scripts
- Mature proof assistants support both directions
- Declarative versus imperative
  - imperative:
    - better for finding proofs (can invoke proof search engines)
    - provides better API for embedded proof
  - declarative:
    - more natural for checking textbook like proofs
    - but rather verbose and 'COBOL like'
- Mature proof assistants support both declarative and imperative styles

A common verification scenario

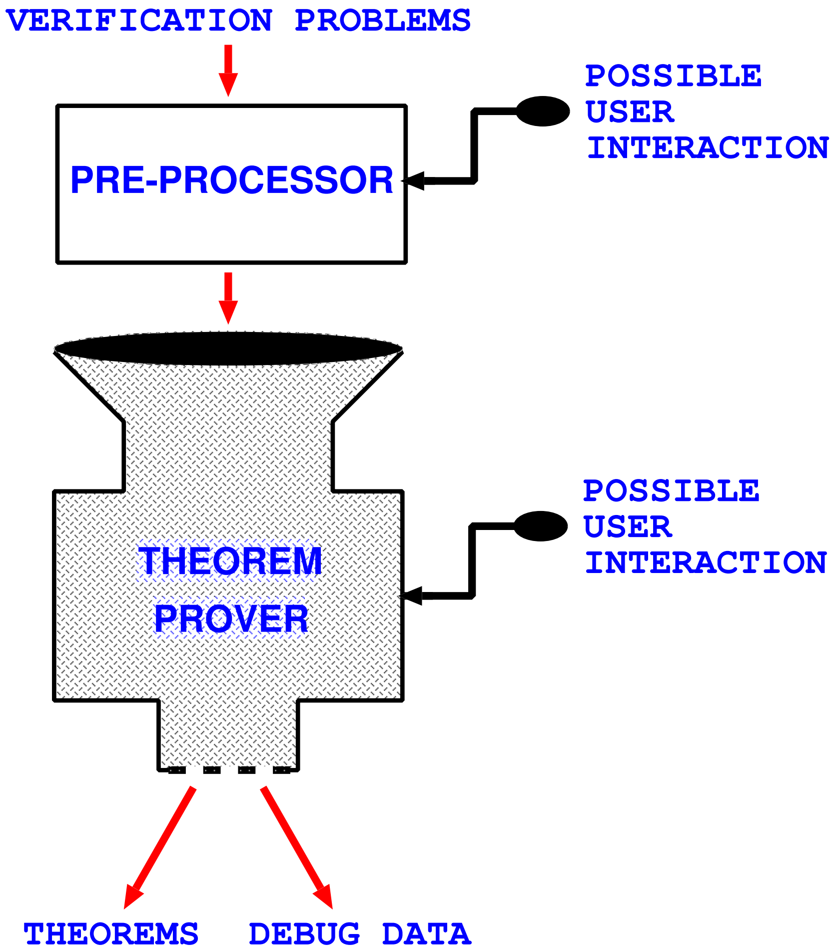


A common verification scenario



- Input verification problem
- Generate proof problem
- Solve with theorem prover

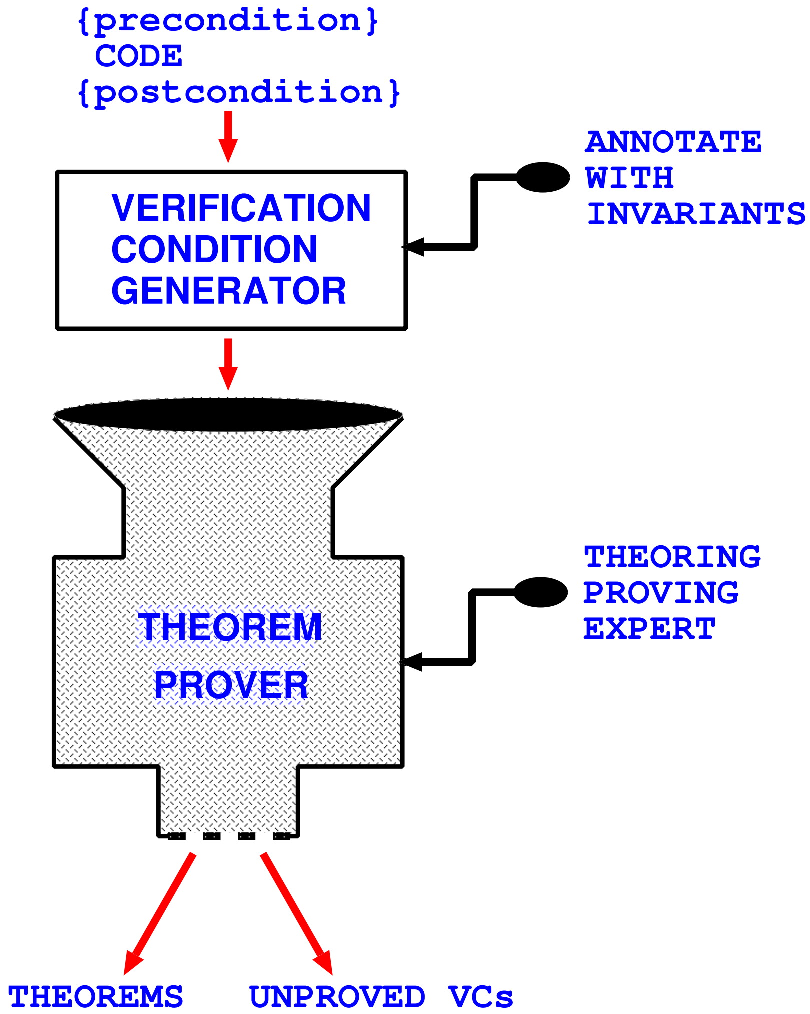
A common verification scenario



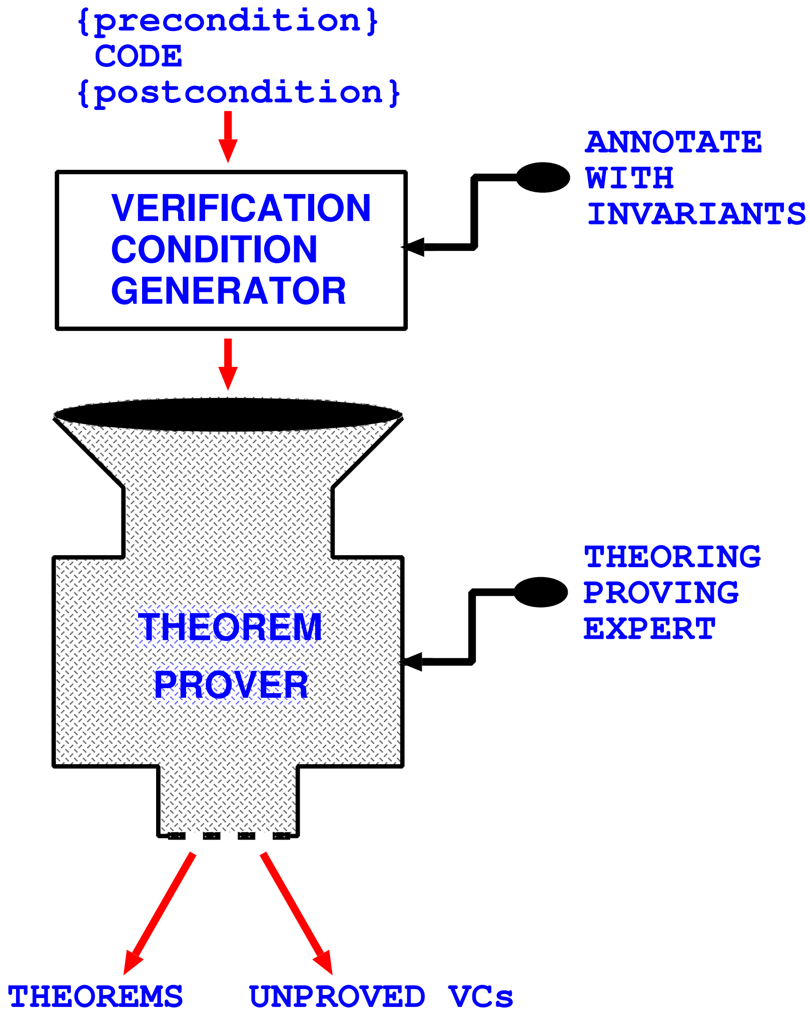
- Input verification problem
- Generate proof problem
- Solve with theorem prover

- Ideally no user interaction
- Pre-processor may need extra input (e.g. pragmas, annotations)
- Prover may need lots of help

Example 1

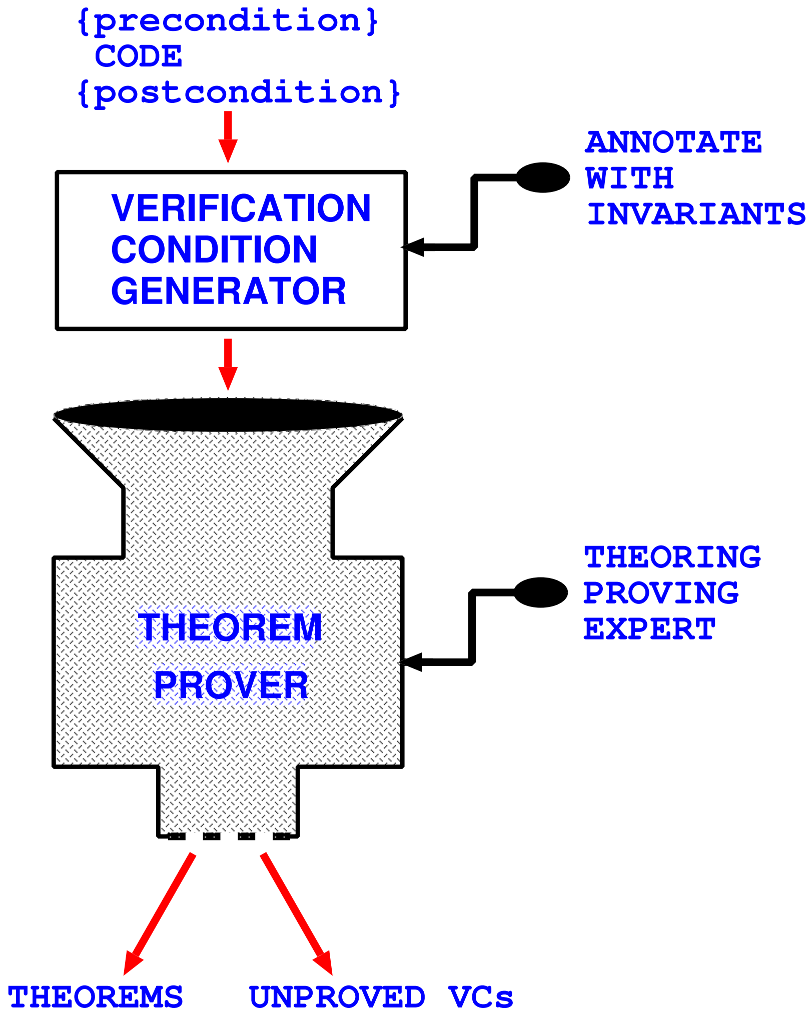

{precondition}
CODE
{postcondition}
VERIFICATION
CONDITION
GENERATOR
ANNOTATE
WITH
INVARIANTS
THEOREM
PROVER
THEORING
PROVING
EXPERT
THEOREMS
UNPROVED VCs

Example 1



- Input code + pre and post conditions
- Compute VCs
  - user adds annotations
  - then VCs generated automatically
- VCs fed to a theorem prover

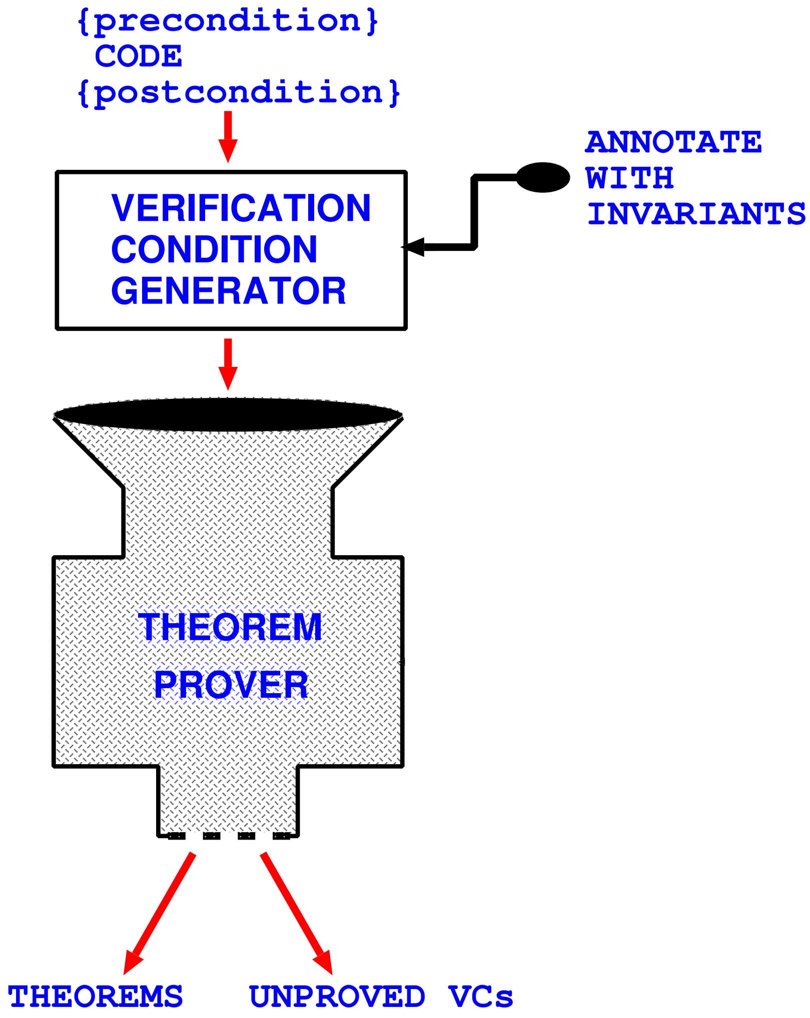
Example 1



- Input code + pre and post conditions
- Compute VCs
  - user adds annotations
  - then VCs generated automatically
- VCs fed to a theorem prover

- Classical program verification
  - Gypsy
  - Stanford Pascal Verifier

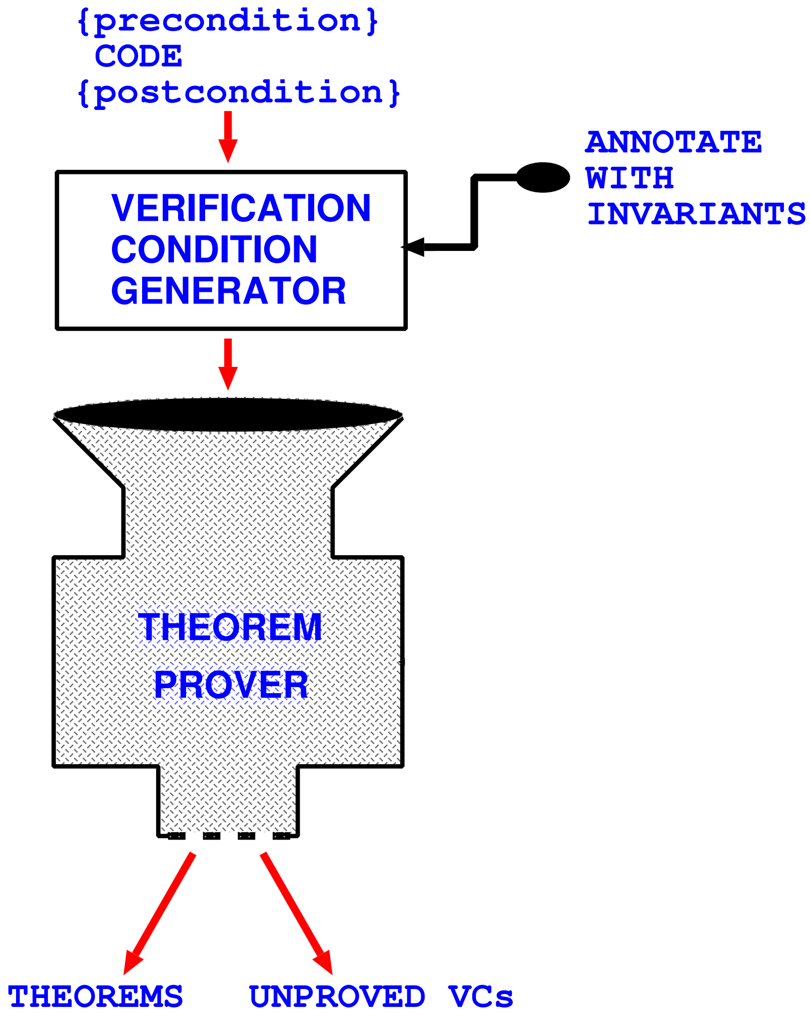
Example 1



- Input code + pre and post conditions
- Compute VCs
  - user adds annotations
  - then VCs generated automatically
- VCs fed to a theorem prover

- Classical program verification
  - Gypsy
  - Stanford Pascal Verifier
- Extended static checking (ESC)

Example 1

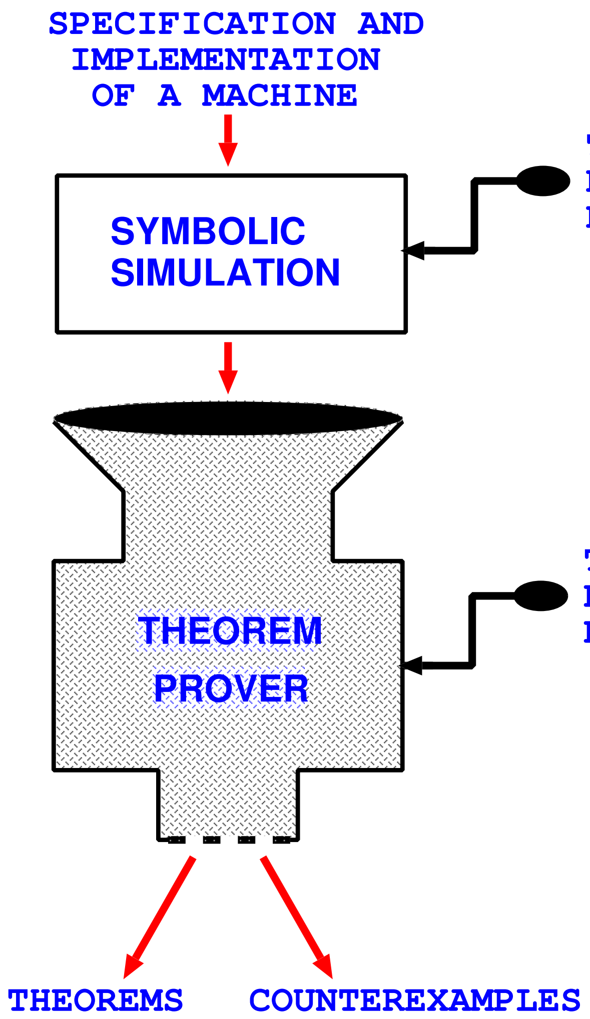


- Input code + pre and post conditions
- Compute VCs
  - user adds annotations
  - then VCs generated automatically
- VCs fed to a theorem prover

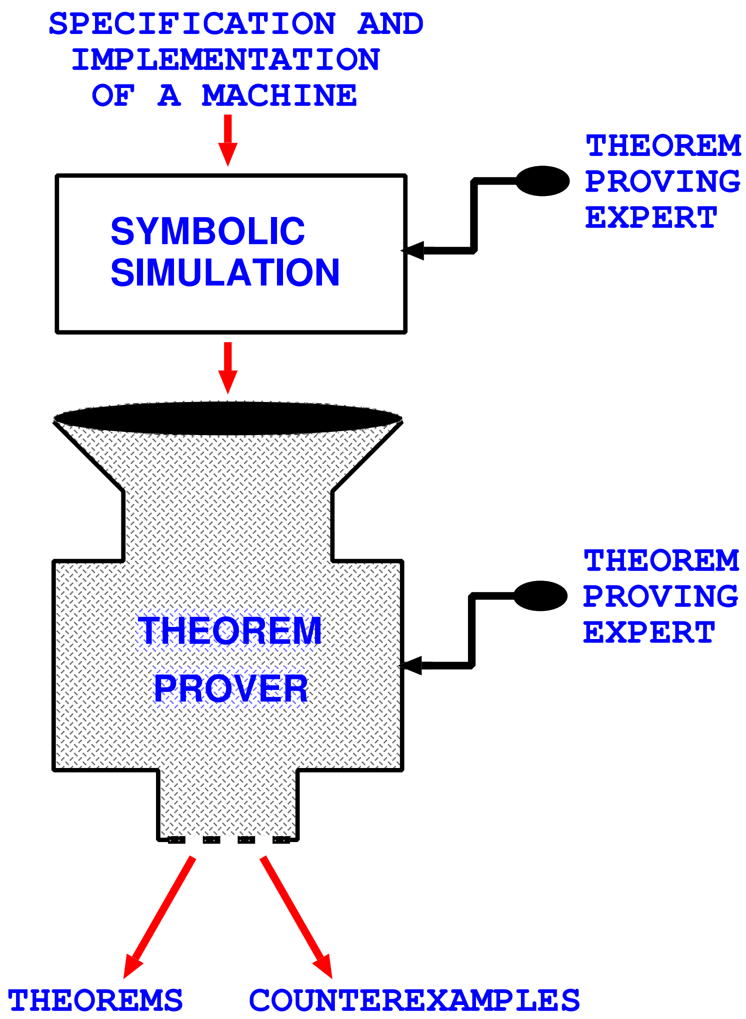
. . . .

- Classical program verification
  - Gypsy
  - Stanford Pascal Verifier
- Extended static checking (ESC)
- Maybe basis for deeper ABV?

Example 2

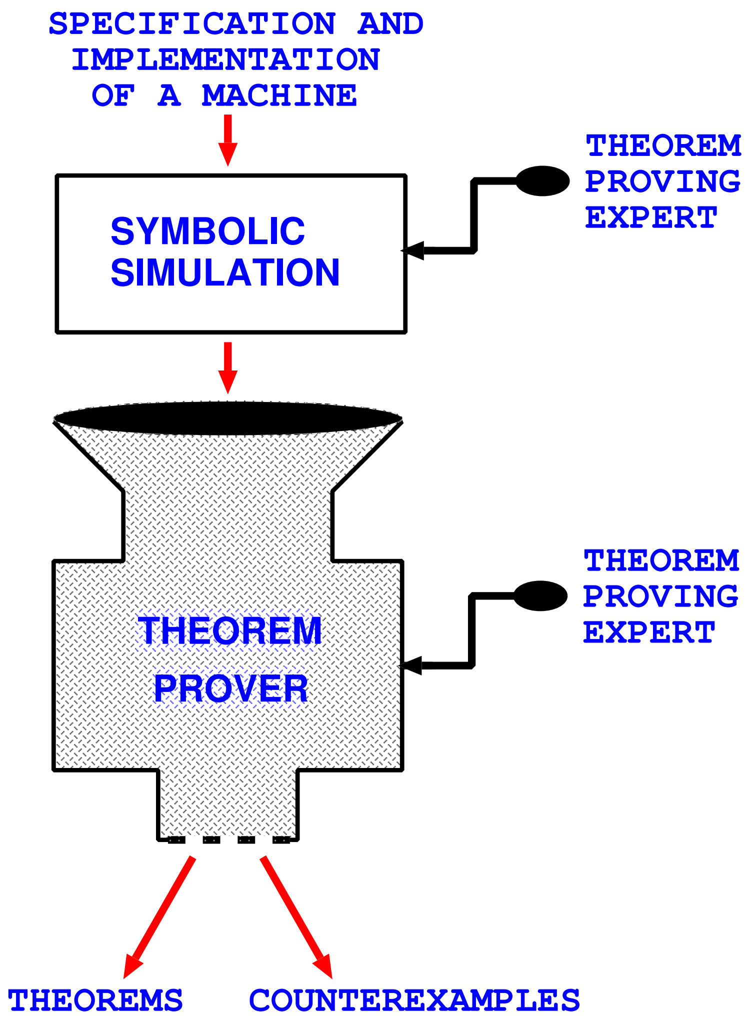

SPECIFICATION AND
IMPLEMENTATION
OF A MACHINE
SYMBOLIC
SIMULATION
THEOREM
PROVING
EXPERT
THEOREM
PROVER
THEOREM
PROVING
EXPERT
THEOREMS
COUNTEREXAMPLES

Example 2



- Input machines
  - specification transition function
  - implementation transition function
- Simulate to matching states
- Prove equivalence

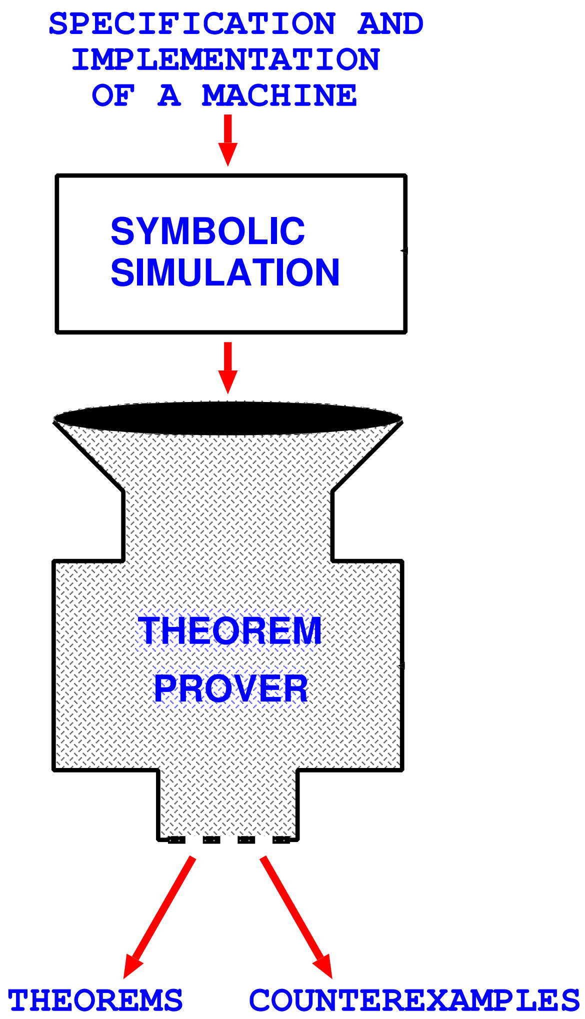
Example 2



- Input machines
  - specification transition function
  - implementation transition function
- Simulate to matching states
- Prove equivalence

- Year long processor verifications
  - ACL2, HOL, PVS
  - symbolic simulation by proof

Example 2

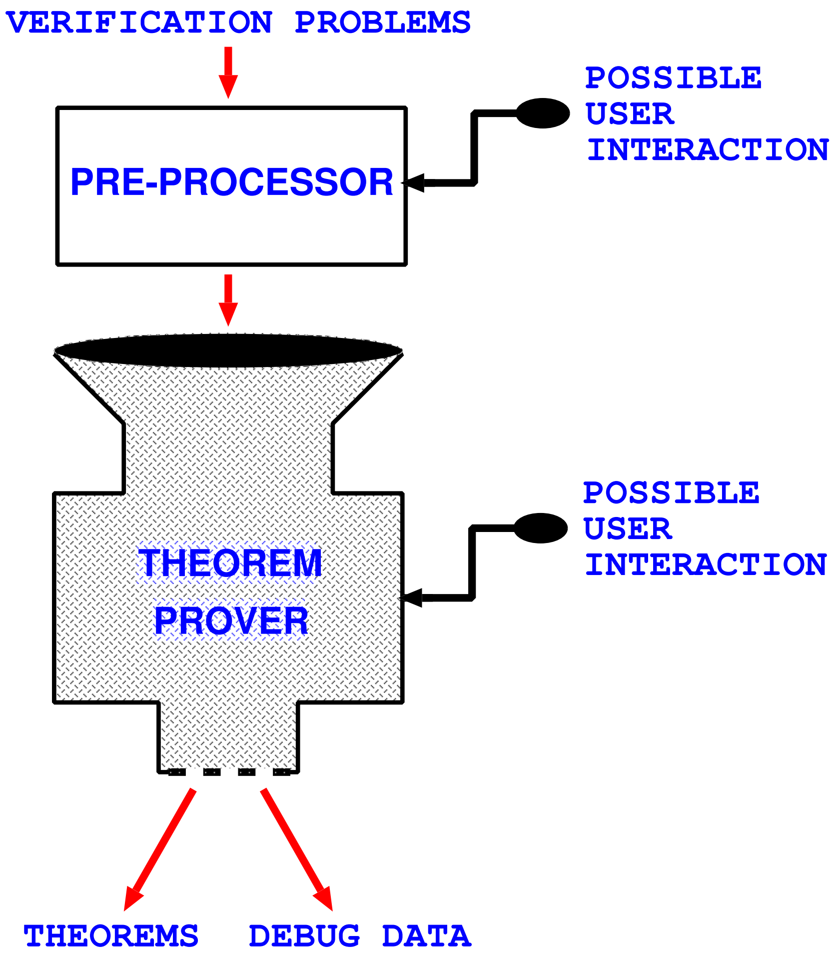


- Input machines
  - specification transition function
  - implementation transition function
- Simulate to matching states
- Prove equivalence

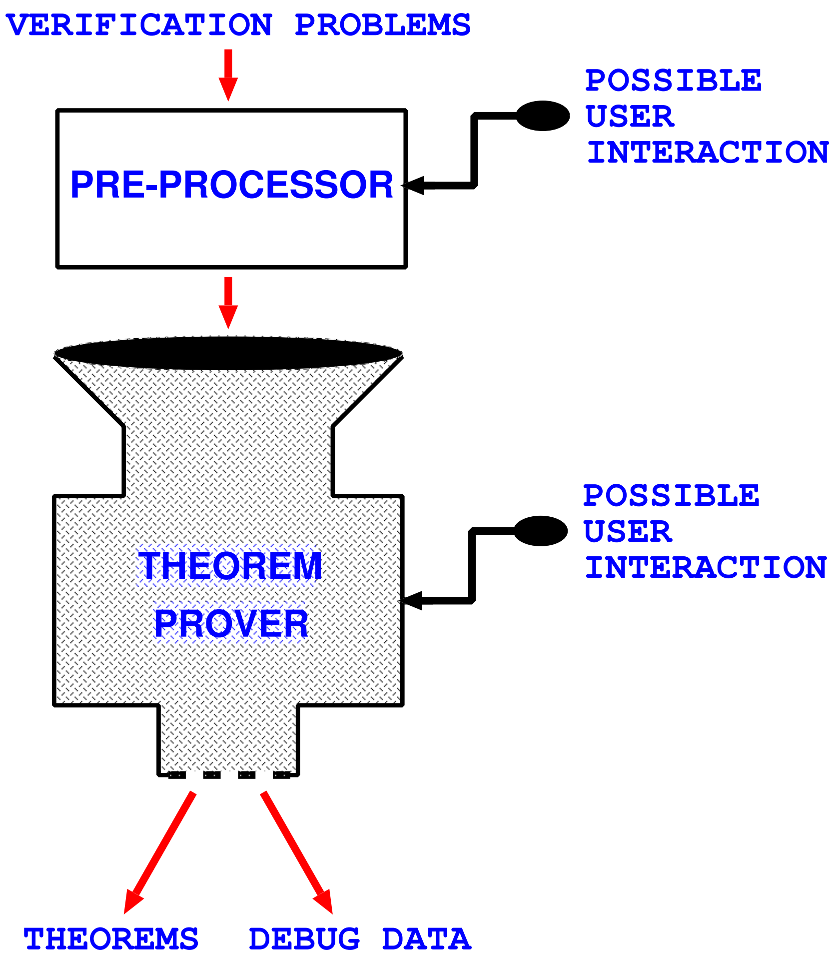
. . . .

- Year long processor verifications
  - ACL2, HOL, PVS
  - symbolic simulation by proof
- Automatic pipeline verification
  - e.g. Burch and Dill
  - symbolic simulation not by proof

Methodology issues

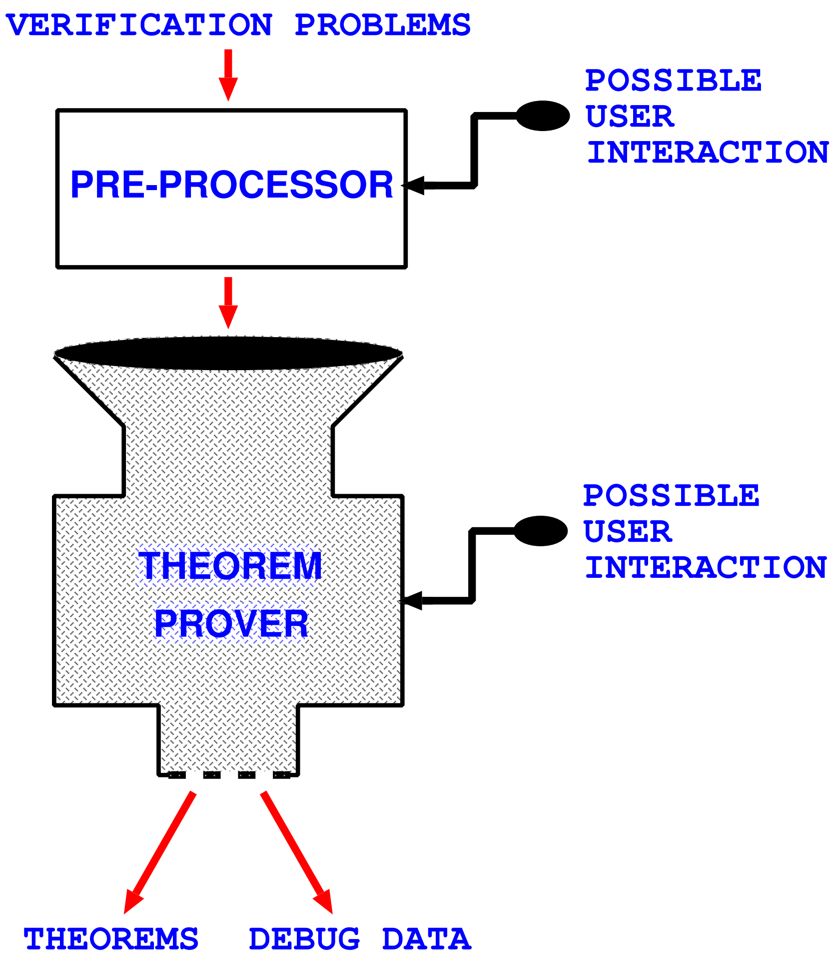


Methodology issues



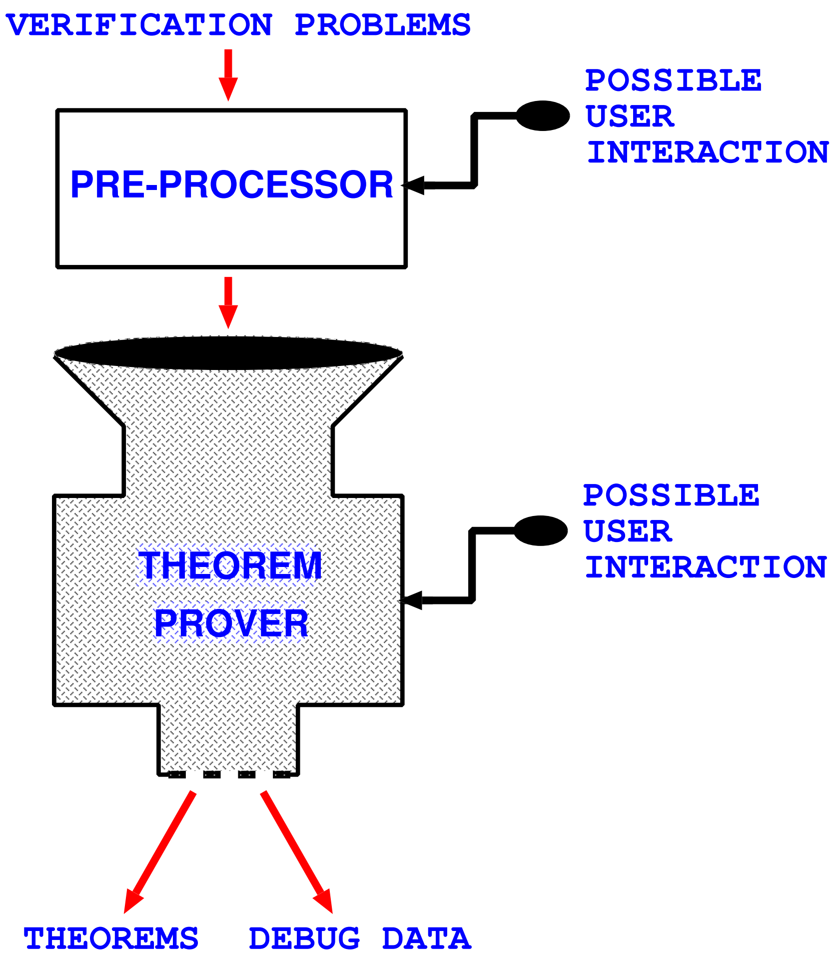
- What problem description language?
- How is pre-processing implemented?
- What kind of theorem prover?
- How are bits glued together?

Methodology issues: Extreme 1



- What problem description language?
  - problem-specific language
- How is pre-processing implemented?
  - pre-processing is YACC
- What kind of theorem prover?
  - problem-specific algorithm
- How are bits glued together?
  - glue is scripting in C, Perl etc.

Methodology issues: ~~Extreme 1~~ Extreme 2



- What problem description language?
  - ~~problem specific language~~
  - problem represented in a logic
- How is pre-processing implemented?
  - ~~pre-processing is YACC~~
  - pre-processing is rewriting
- What kind of theorem prover?
  - ~~problem specific algorithm~~
  - general purpose prover
- How are bits glued together?
  - ~~glue is scripting in C, Perl etc.~~
  - glue is formal proof

Issues

- What problem description language?
- Is pre-processing formal?
- What kind of theorem prover?
- How are bits glued together?

Issues .... Extreme 1

- Extreme 1
  - efficient, scary

- What problem description language?
  1. ad-hoc problem language
- Is pre-processing formal?
  1. pre-processing is YACC
- What kind of theorem prover?
  1. problem-specific algorithm
- How are bits glued together?
  1. glue is scripting: C, Perl etc.

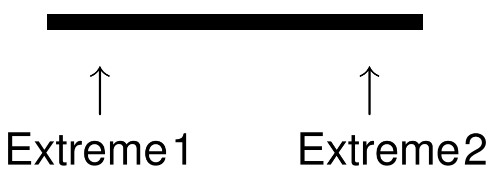
Issues .... Extreme 1 .... Extreme 2

- Extreme 1
  - efficient, scary
- Extreme 2
  - inefficient, reassuring

- What problem description language?
  1. ad-hoc problem language
  2. problem represented in a logic
- Is pre-processing formal?
  1. pre-processing is YACC
  2. pre-processing is formal proof
- What kind of theorem prover?
  1. problem-specific algorithm
  2. general purpose prover
- How are bits glued together?
  1. glue is scripting: C, Perl etc.
  2. glue is formal proof

Issues .... Extreme 1 .... Extreme 2

- Extreme 1
  - efficient, scary
- Extreme 2
  - inefficient, reassuring
- Spectrum:



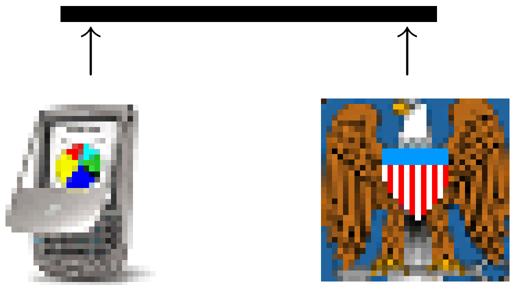
- What problem description language?
  1. ad-hoc problem language
  2. problem represented in a logic
- Is pre-processing formal?
  1. pre-processing is YACC
  2. pre-processing is formal proof
- What kind of theorem prover?
  1. problem-specific algorithm
  2. general purpose prover
- How are bits glued together?
  1. glue is scripting: C, Perl etc.
  2. glue is formal proof

Programming the spectrum of provers

- Need both efficiency and soundness

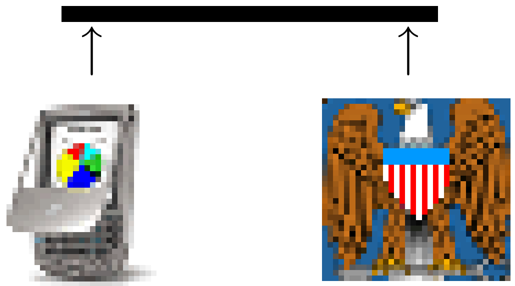
Programming the spectrum of provers

- Need both efficiency and soundness
  - balance depends on application



Programming the spectrum of provers

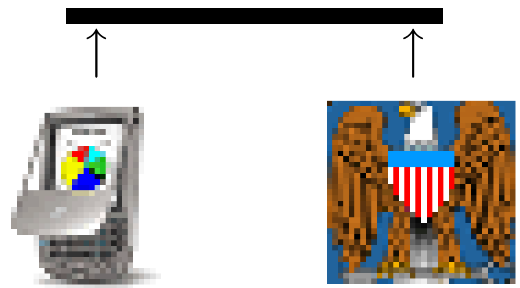
- Need both efficiency and soundness
  - balance depends on application



- Would like IDE to help manage programming across spectrum

Programming the spectrum of provers

- Need both efficiency and soundness
  - balance depends on application



- Would like IDE to help manage programming across spectrum
  - IDE: Integrated Development Environment
  - provides proof engine components
  - and ways of linking them

Proof engines one might want to combine

- Little engines of proof . . . . (*cf.* Shankar, FLoC'02)
  - automatic
- Medium engines of proof
  - mainly automatic, lightweight user guidance
- Big engines of proof
  - user guided, but may have automatic tools (smaller engines)

Proof engines one might want to combine

- Little engines of proof . . . . (*cf.* Shankar, FLoC'02)
  - automatic
    - propositional solvers (SAT)
    - decision procedures, model checkers (arithmetic, LTL, CTL, M2L)
- Medium engines of proof
  - mainly automatic, lightweight user guidance
- Big engines of proof
  - user guided, but may have automatic tools (smaller engines)

Proof engines one might want to combine

- Little engines of proof . . . . (*cf.* Shankar, FLoC'02)
  - automatic
    - propositional solvers (SAT)
    - decision procedures, model checkers (arithmetic, LTL, CTL, M2L)
- Medium engines of proof
  - mainly automatic, lightweight user guidance
    - model checkers with lightweight proof (SMV, Forte)
    - first-order provers (SVC/CVC, EVC, Gandalf, Otter, SPASS, INKA, Vampire)
- Big engines of proof
  - user guided, but may have automatic tools (smaller engines)

Proof engines one might want to combine

- Little engines of proof .... (*cf.* Shankar, FLoC'02)
  - automatic
    - propositional solvers (SAT)
    - decision procedures, model checkers (arithmetic, LTL, CTL, M2L)
- Medium engines of proof
  - mainly automatic, lightweight user guidance
    - model checkers with lightweight proof (SMV, Forte)
    - first-order provers (SVC/CVC, EVC, Gandalf, Otter, SPASS, INKA, Vampire)
- Big engines of proof
  - user guided, but may have automatic tools (smaller engines)
    - specific applications (LAMBDA, STeP)
    - general (PVS, Isabelle, HOL, ProofPower, Acl2, Nuprl, OMEGA, Eves, IMPS, Coq)

Combining ‘engines of proof’

- Middleware exists for connecting tools

Combining ‘engines of proof’

- Middleware exists for connecting tools
  - Prosper, MathWeb Software Bus, XML
  - rather heavy, still experimental

Combining ‘engines of proof’

- Middleware exists for connecting tools
  - Prosper, MathWeb Software Bus, XML
  - rather heavy, still experimental
- However, big engines of proof have evolved to be proof IDEs

Combining ‘engines of proof’

- Middleware exists for connecting tools
  - Prosper, MathWeb Software Bus, XML
  - rather heavy, still experimental
- However, big engines of proof have evolved to be proof IDEs
  - support efficient execution

Combining ‘engines of proof’

- Middleware exists for connecting tools
  - Prosper, MathWeb Software Bus, XML
  - rather heavy, still experimental
- However, big engines of proof have evolved to be proof IDEs
  - support efficient execution
  - have scripting language
    - usually based on Lisp or ML

Combining 'engines of proof'

- Middleware exists for connecting tools
  - Prosper, MathWeb Software Bus, XML
  - rather heavy, still experimental
- However, big engines of proof have evolved to be proof IDEs
  - support efficient execution
  - have scripting language
    - usually based on Lisp or ML
  - can invoke smaller engines of proof as components
    - model checkers, SAT provers etc.

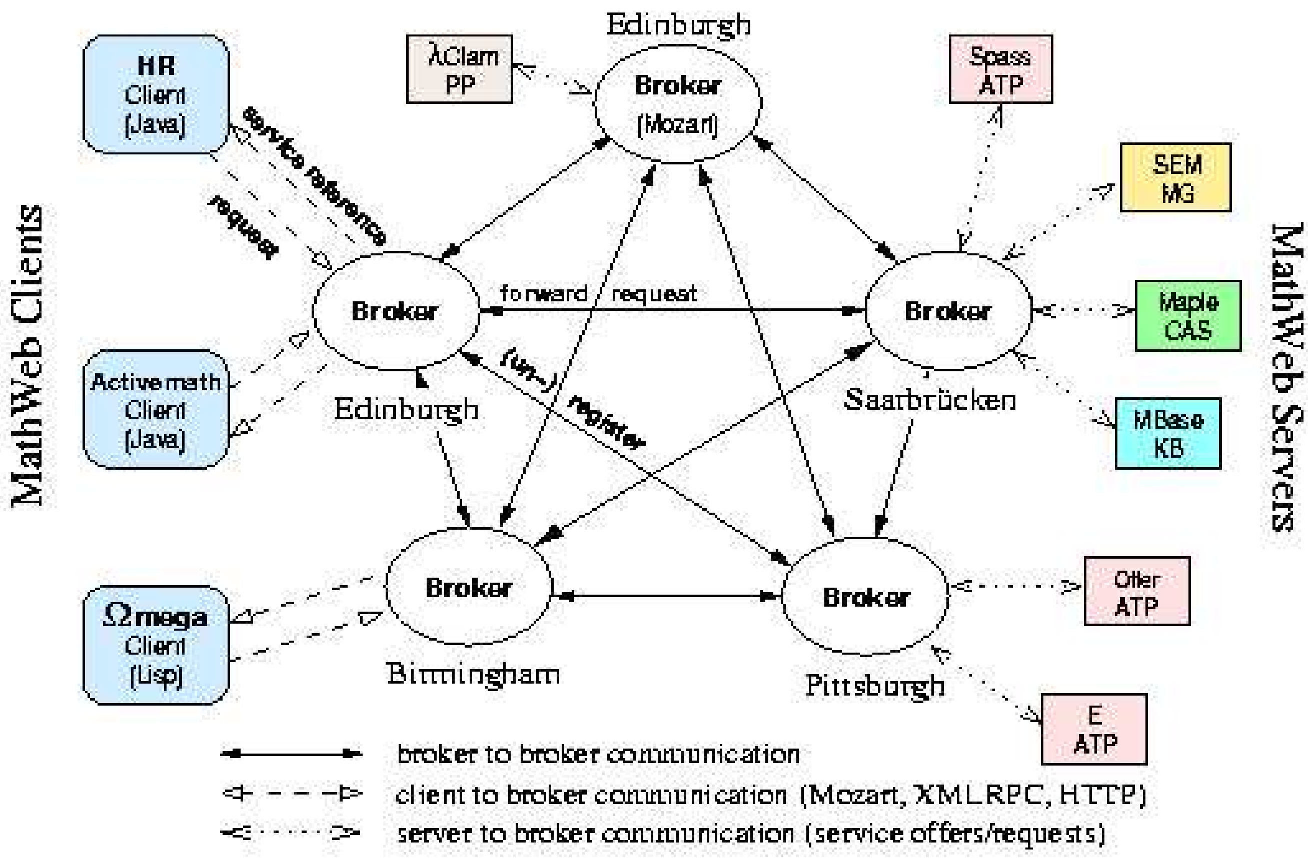
Combining 'engines of proof'

- Middleware exists for connecting tools
  - Prosper, MathWeb Software Bus, XML
  - rather heavy, still experimental
- However, big engines of proof have evolved to be proof IDEs
  - support efficient execution
  - have scripting language
    - usually based on Lisp or ML
  - can invoke smaller engines of proof as components
    - model checkers, SAT provers etc.
  - can do fine grain programming of sequences of inference steps

Combining ‘engines of proof’

- Middleware exists for connecting tools
  - Prosper, MathWeb Software Bus, XML
  - rather heavy, still experimental
- However, big engines of proof have evolved to be proof IDEs
  - support efficient execution
  - have scripting language
    - usually based on Lisp or ML
  - can invoke smaller engines of proof as components
    - model checkers, SAT provers etc.
  - can do fine grain programming of sequences of inference steps
    - maybe using efficient representations (e.g. BDD operations)

MathWeb (http://www.mathweb.org/mathweb/demo.html)



Theorem prover as tool implementation platform

- Not a new idea:
  - 1973. Kowalski & Hayes, Colmerauer: logic programming
  - 1998. CAV 98: Hardin/Wilding/Greve:
    "Transforming the Theorem Prover into a Digital Design Tool"

Theorem prover as tool implementation platform

- Not a new idea:
  - 1973. Kowalski & Hayes, Colmerauer: logic programming
  - 1998. CAV 98: Hardin/Wilding/Greve:
    "Transforming the Theorem Prover into a Digital Design Tool"
- Current big theorem provers are result of 30 years of evolution

Theorem prover as tool implementation platform

- Not a new idea:
  - 1973. Kowalski & Hayes, Colmerauer: logic programming
  - 1998. CAV 98: Hardin/Wilding/Greve:
    "Transforming the Theorem Prover into a Digital Design Tool"
- Current big theorem provers are result of 30 years of evolution
- Issue: link engines via middleware or invoke them from a proof IDE?

Theorem prover as tool implementation platform

- Not a new idea:
  - 1973. Kowalski & Hayes, Colmerauer: logic programming
  - 1998. CAV 98: Hardin/Wilding/Greve:
    "Transforming the Theorem Prover into a Digital Design Tool"
- Current big theorem provers are result of 30 years of evolution
- Issue: link engines via middleware or invoke them from a proof IDE?
- My opinions

Theorem prover as tool implementation platform

- Not a new idea:
  - 1973. Kowalski & Hayes, Colmerauer: logic programming
  - 1998. CAV 98: Hardin/Wilding/Greve:
    "Transforming the Theorem Prover into a Digital Design Tool"
- Current big theorem provers are result of 30 years of evolution
- Issue: link engines via middleware or invoke them from a proof IDE?
- My opinions
  - middleware experiments (e.g. Prosper) have raised issues
    - efficient data exchange, controlling (starting/stopping engines)
  - proof engines need good APIs
  - CORBA type solution too heavy
  - theorem prover as IDE is manageable

Theorem prover as tool implementation platform

- Not a new idea:
  - 1973. Kowalski & Hayes, Colmerauer: logic programming
  - 1998. CAV 98: Hardin/Wilding/Greve:
    "Transforming the Theorem Prover into a Digital Design Tool"
- Current big theorem provers are result of 30 years of evolution
- Issue: link engines via middleware or invoke them from a proof IDE?
- My opinions
  - middleware experiments (e.g. Prosper) have raised issues
    - efficient data exchange, controlling (starting/stopping engines)
  - proof engines need good APIs
  - CORBA type solution too heavy
  - theorem prover as IDE is manageable . . . . and available now!

Debugging versus correctness

- Formal verification can be viewed as debugging

- Ultimate goal is proof of correctness

Debugging versus correctness

- Formal verification can be viewed as debugging
  - fits into standard verification flow
    - FV can start from states produced by simulation
    - gets hard-to-find bugs
  - but how does one know when to stop debugging?
- Ultimate goal is proof of correctness

Debugging versus correctness

- Formal verification can be viewed as debugging
  - fits into standard verification flow
    - FV can start from states produced by simulation
    - gets hard-to-find bugs
  - but how does one know when to stop debugging?
- Ultimate goal is proof of correctness
  - provides much stronger assurance that everything is covered
    - coverage assurance only as good as the specification
  - widely thought to be impractical

Debugging versus correctness

- Formal verification can be viewed as debugging
  - fits into standard verification flow
    - FV can start from states produced by simulation
    - gets hard-to-find bugs
  - but how does one know when to stop debugging?
- Ultimate goal is proof of correctness
  - provides much stronger assurance that everything is covered
    - coverage assurance only as good as the specification
  - widely thought to be impractical
    - that was 1990s, is practical now . . . . though expensive

Opinions are divided: recent quotes found on the web

- Find bugs, not proofs

- Real value is assurance that there are no bugs

Opinions are divided: recent quotes found on the web

- 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

Opinions are divided: recent quotes found on the web

- 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

Opinions are divided: recent quotes found on the web

- 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

- Counter-example technologies are a stepping stone
- Finding bugs has immediate value, but proof can deliver much more
- Full correctness assurance is possible now, and the cost is falling!

My opinions

- Counter-example technologies are a stepping stone

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

- Finding bugs has immediate value, but proof can deliver much more

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

My opinions

- Counter-example technologies are a stepping stone

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

- Finding bugs has immediate value, but proof can deliver much more
  - gets foot in the door for full formal specification
  - need more traditional/formal combinations .... more feet in door

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

My opinions

- Counter-example technologies are a stepping stone

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

- Finding bugs has immediate value, but proof can deliver much more
  - gets foot in the door for full formal specification
  - need more traditional/formal combinations .... more feet in door

- 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
  - components need specifications and correct implementations

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?
  - design IP needs multilevel specifications (RTL, behavioral)
  - specifications are more valuable if correct
  - 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

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
- Could also produce custom theorem proving environments
  - synthesise processor & hardware/software FV tools from specification
  - generate bespoke ESL/co-design theorem proving environments

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.

⋮

Conclusions

- Issues

- Challenges

Conclusions

- Issues
  - declarative versus imperative; forward versus backward
  - efficiency versus coherence (CORBA vs. proof IDE)
  - plethora of logics
  - user-interface versus API
  - counter-examples versus proof of correctness
- Challenges

Conclusions

- Issues
  - declarative versus imperative; forward versus backward
  - efficiency versus coherence (CORBA vs. proof IDE)
  - plethora of logics
  - user-interface versus API
  - counter-examples versus proof of correctness
- Challenges
  - getting theorem proving into real verification flows
  - advance state-of-the-art
  - keep up demonstrator projects
  - make a market for specification and proof IP

Conclusions

- Issues
  - declarative versus imperative; forward versus backward
  - efficiency versus coherence (CORBA vs. proof IDE)
  - plethora of logics
  - user-interface versus API
  - counter-examples versus proof of correctness
- Challenges
  - getting theorem proving into real verification flows
    - Intel, AMD can do it; tough for small companies
    - proof engine deployment platform: *ProofStudio.net*
  - advance state-of-the-art
  - keep up demonstrator projects
  - make a market for specification and proof IP

Conclusions

- Issues
  - declarative versus imperative; forward versus backward
  - efficiency versus coherence (CORBA vs. proof IDE)
  - plethora of logics
  - user-interface versus API
  - counter-examples versus proof of correctness
- Challenges
  - getting theorem proving into real verification flows
    - Intel, AMD can do it; tough for small companies
    - proof engine deployment platform: *ProofStudio.net*
  - advance state-of-the-art
    - better integrate first order reasoning, equality & decision procedures
    - go beyond first order automation
  - keep up demonstrator projects
  - make a market for specification and proof IP

Conclusions

- Issues
  - declarative versus imperative; forward versus backward
  - efficiency versus coherence (CORBA vs. proof IDE)
  - plethora of logics
  - user-interface versus API
  - counter-examples versus proof of correctness
- Challenges
  - getting theorem proving into real verification flows
    - Intel, AMD can do it; tough for small companies
    - proof engine deployment platform: *ProofStudio.net*
  - advance state-of-the-art
    - better integrate first order reasoning, equality & decision procedures
    - go beyond first order automation
  - keep up demonstrator projects
    - hard to motivate as ‘proof of concept’ established
    - need more cost/benefit stories
  - make a market for specification and proof IP

Conclusions .... Long Live Theorem Proving!

- Issues
  - declarative versus imperative; forward versus backward
  - efficiency versus coherence (CORBA vs. proof IDE)
  - plethora of logics
  - user-interface versus API
  - counter-examples versus proof of correctness
- Challenges
  - getting theorem proving into real verification flows
    - Intel, AMD can do it; tough for small companies
    - proof engine deployment platform: *ProofStudio.net*
  - advance state-of-the-art
    - better integrate first order reasoning, equality & decision procedures
    - go beyond first order automation
  - keep up demonstrator projects
    - hard to motivate as 'proof of concept' established
    - need more cost/benefit stories
  - make a market for specification and proof IP

THE END

Emergency slides for if I finish too early!

-

-

Emergency slides for if I finish too early!

- Some applications and spinoff from theorem proving research
-

Emergency slides for if I finish too early!

- Some applications and spinoff from theorem proving research
- Quotes from the web

Two applications of theorem proving research

- Tenison VTOC^{TM}

- Processing semantics of Accellera PSL/Sugar property language

- Features of these examples

- Opinion

Two applications of theorem proving research

- Tenison VTOC^{TM}
  - new commercial product
  - not FV, but exploits ideas from semantics and theorem proving
- Processing semantics of Accellera PSL/Sugar property language
- Features of these examples
- Opinion

Two applications of theorem proving research

- Tenison VTOC™
  - new commercial product
  - not FV, but exploits ideas from semantics and theorem proving
- Processing semantics of Accellera PSL/Sugar property language
  - current research at Cambridge
  - use a theorem prover to generate verification tools
- Features of these examples
- Opinion

Two applications of theorem proving research

- Tenison VTOC^{TM}
  - new commercial product
  - not FV, but exploits ideas from semantics and theorem proving
- Processing semantics of Accellera PSL/Sugar property language
  - current research at Cambridge
  - use a theorem prover to generate verification tools
- Features of these examples
  - use ideas or tools from theorem proving
  - generate tools for standard verification (simulation)
- Opinion

Two applications of theorem proving research

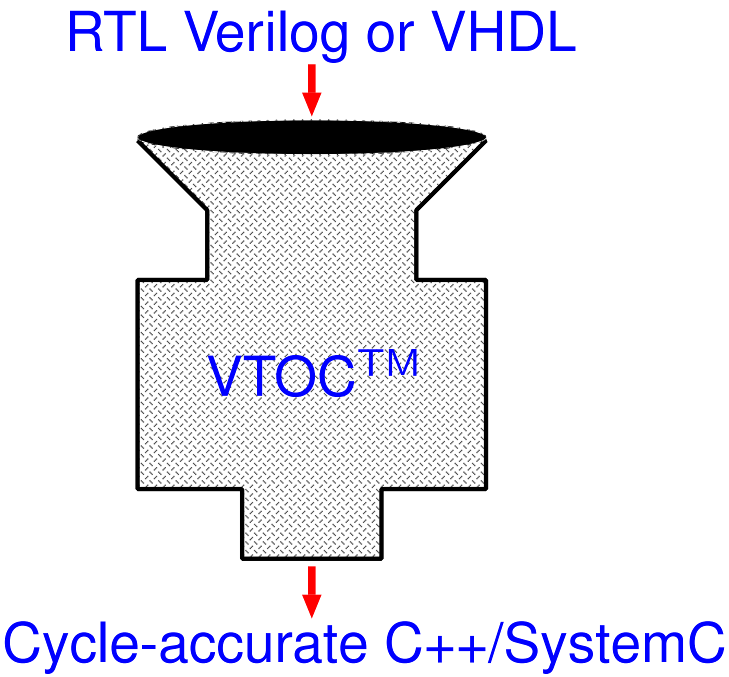
- Tenison VTOC™
  - new commercial product
  - not FV, but exploits ideas from semantics and theorem proving
- Processing semantics of Accellera PSL/Sugar property language
  - current research at Cambridge
  - use a theorem prover to generate verification tools
- Features of these examples
  - use ideas or tools from theorem proving
  - generate tools for standard verification (simulation)
- Opinion
  - need more wacky blue sky research, AI etc.
  - essential investment for long term innovation

Example of spinoff from theorem proving research

- Tenison EDA story (`www.tenison.com`)

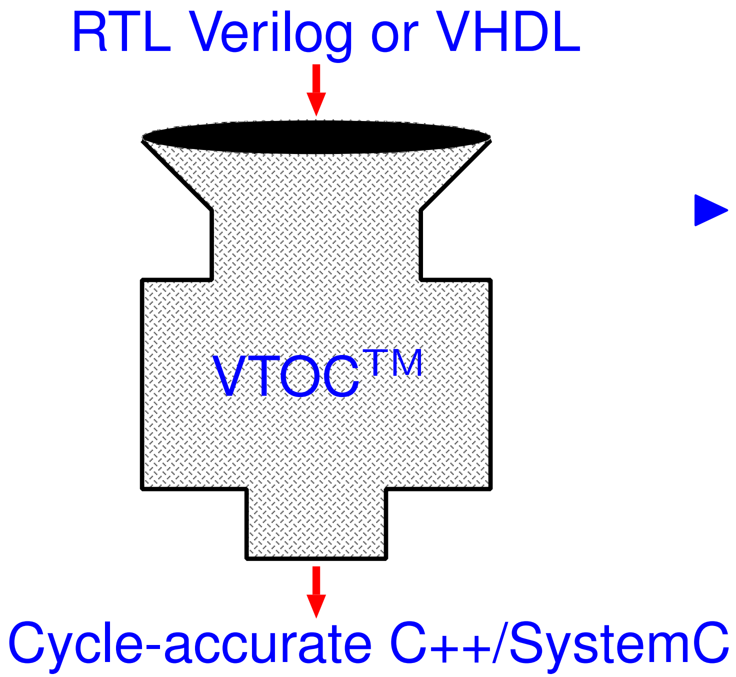
Example of spinoff from theorem proving research

- Tenison EDA story (`www.tenison.com`)



Example of spinoff from theorem proving research

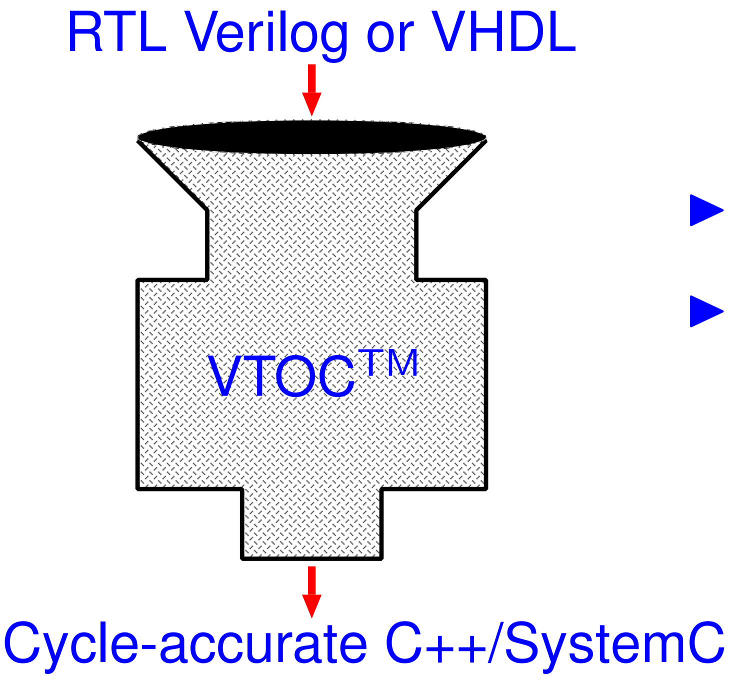
- Tenison EDA story (`www.tenison.com`)



- Programmed in theorem prover language ML

Example of spinoff from theorem proving research

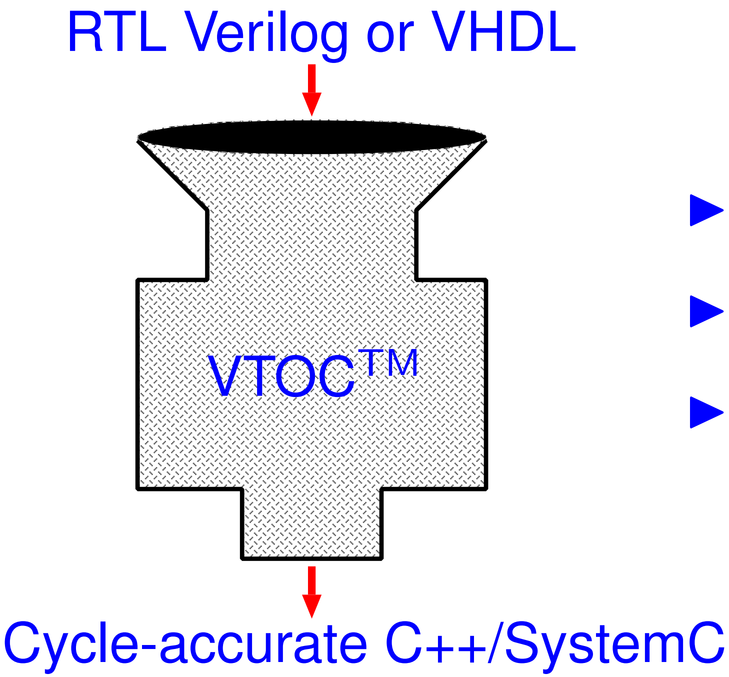
- Tenison EDA story (`www.tenison.com`)



- Programmed in theorem prover language ML
- Based on executing denotational semantics

Example of spinoff from theorem proving research

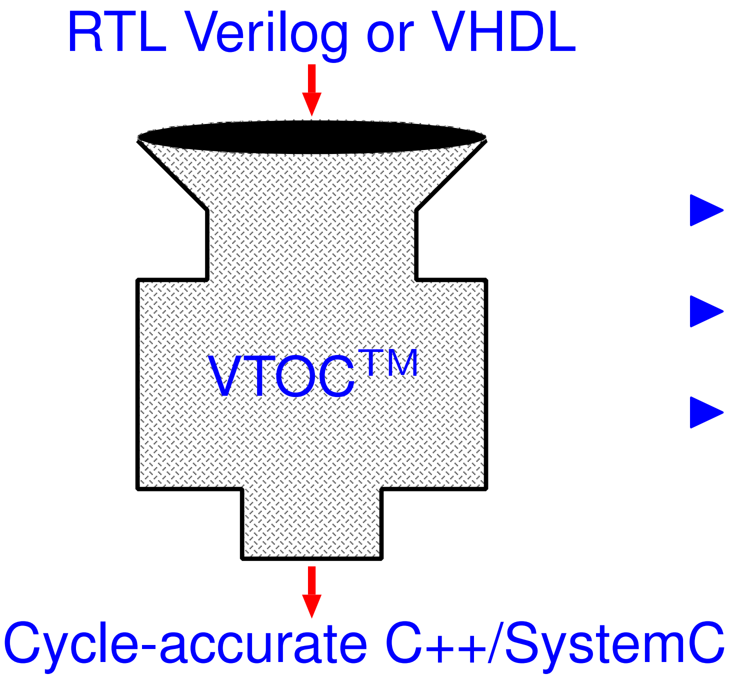
- Tenison EDA story (`www.tenison.com`)



- Programmed in theorem prover language ML
- Based on executing denotational semantics
- Generates very efficient simulation models

Example of spinoff from theorem proving research

- Tenison EDA story (`www.tenison.com`)

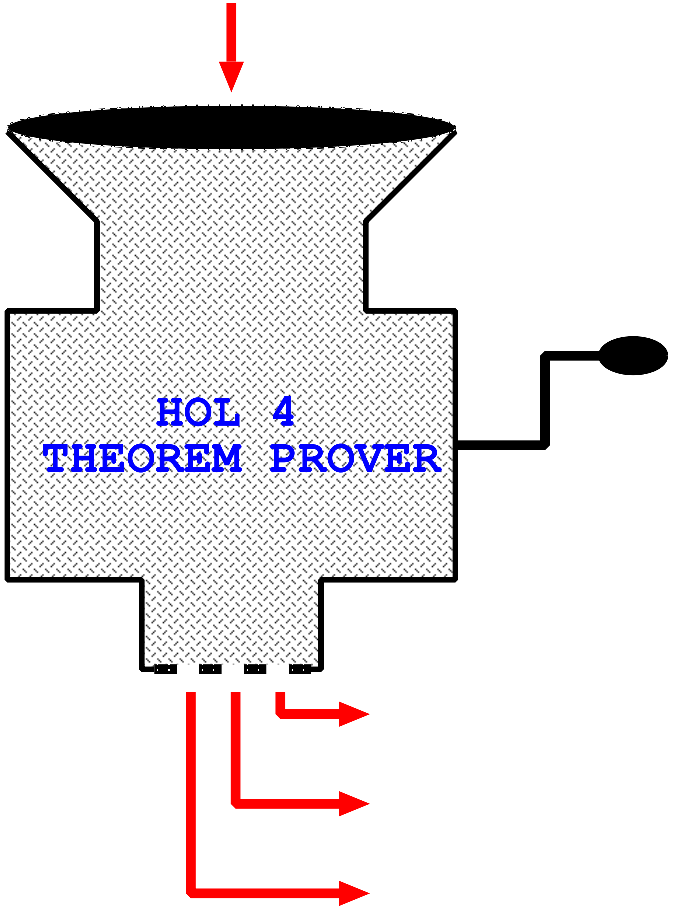


- Programmed in theorem prover language ML
- Based on executing denotational semantics
- Generates very efficient simulation models

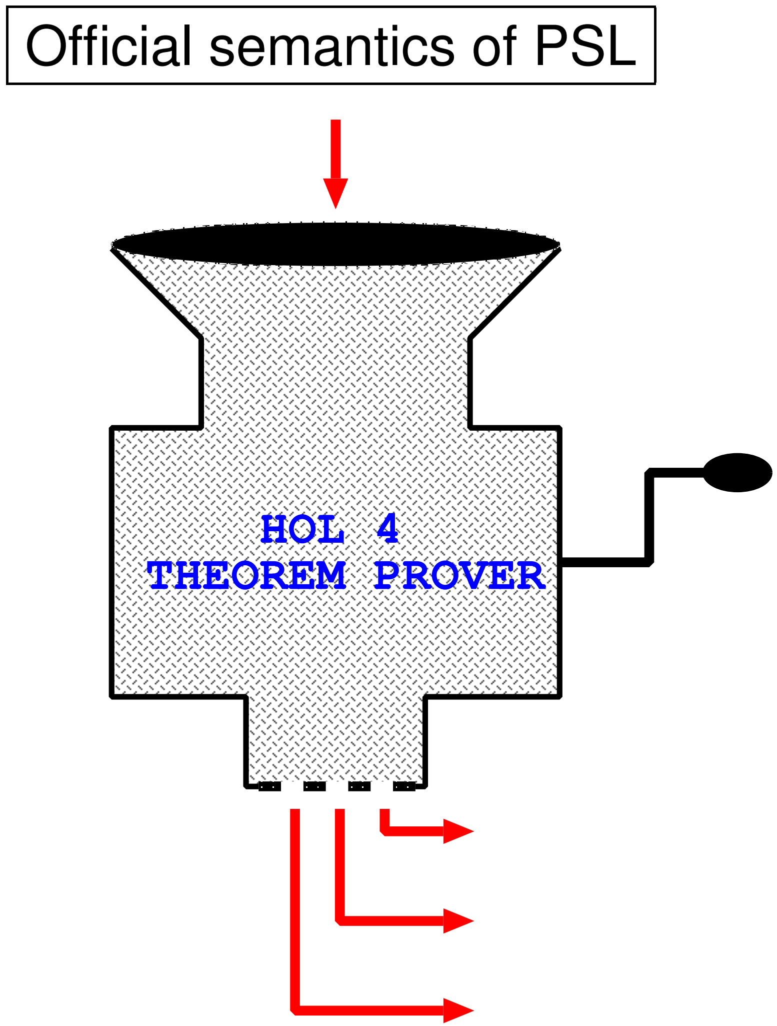
- Moral
  - 'blue sky' research can have unexpected applications
  - a cliché, but still worth repeating

Generating tools for PSL

Official semantics of PSL



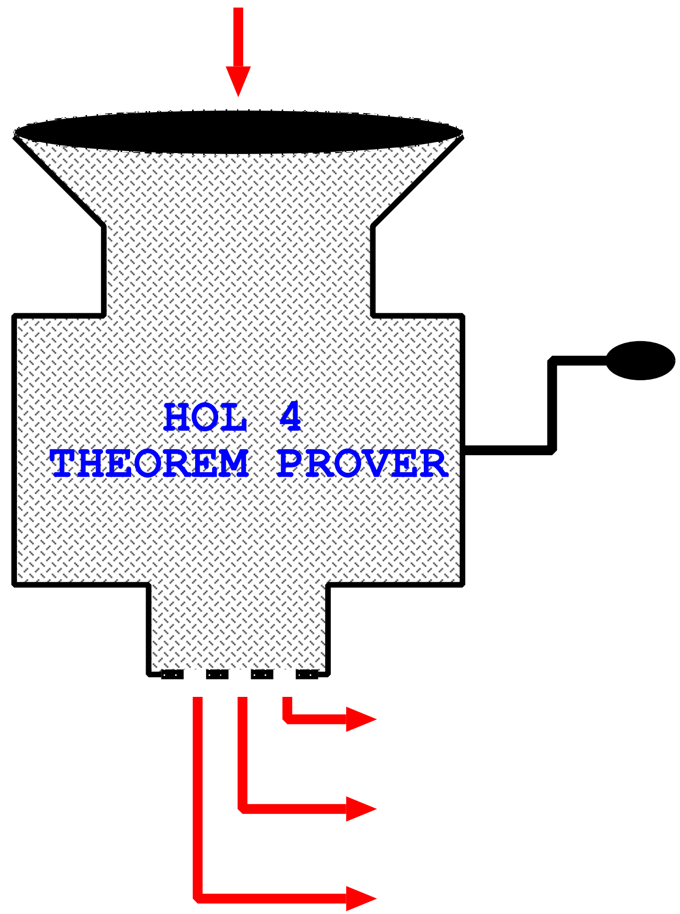
Generating tools for PSL



- Input ‘golden’ semantics from LRM

Generating tools for PSL

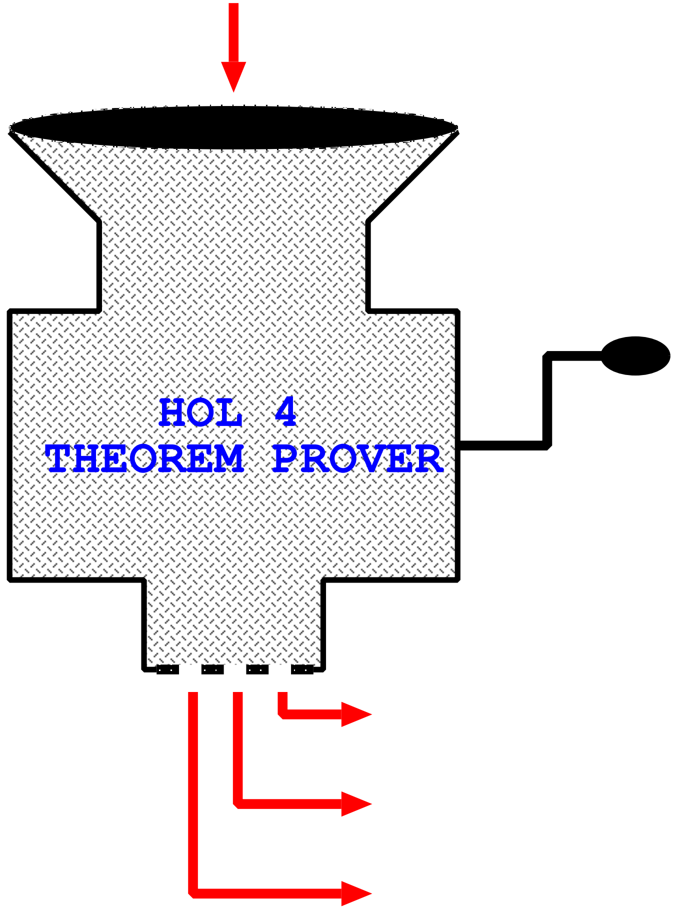
Official semantics of PSL



- Input 'golden' semantics from LRM
- Generate tools by proof

Generating tools for PSL

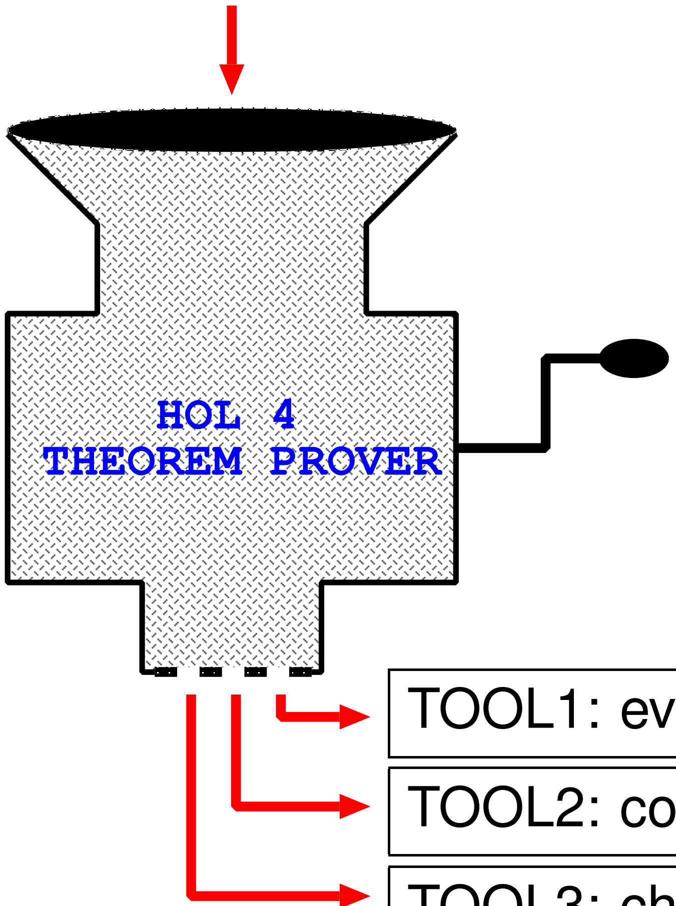
Official semantics of PSL



- Input ‘golden’ semantics from LRM
- Generate tools by proof
- Generated tools work by proof

Generating tools for PSL . . . . Gordon, Hurd, Slind, CHARME 2003

Official semantics of PSL



- Input ‘golden’ semantics from LRM
- Generate tools by proof
- Generated tools work by proof

TOOL1: evaluate LTL properties on a specific path

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

TOOL3: check CTL properties against a model (Amjad’s PhD)

From the web

With re-spin costs of $1 Million or more, and time-to-market a driving concern, how can you be sure that your design is 100% Bug-Free? XXXX™ provides 100% Formal Proof that your design matches Spec-Level Design Requirements ensuring that your design will be right the first time.

From the web

With re-spin costs of $1 Million or more, and time-to-market a driving concern, how can you be sure that your design is 100% Bug-Free? XXXX™ provides 100% Formal Proof that your design matches Spec-Level Design Requirements ensuring that your design will be right the first time.

- Reminds me of *Viper*!

More quotes from the web

. . . a bug which costs $1 to fix on the programmer's desktop costs $100 to fix once it is incorporated into a complete program, and many thousands of dollars if it is identified only after the software has been deployed in the field.

. . . .

However, though formal-methods research may have failed to deliver on the promises of the 1960s, it has still produced a collection of useful techniques. A number of firms are now creating software tools that can allow such techniques to be applied more widely by programmers who are not versed in such formal methods.

The trick is to integrate them into the software systems, called integrated development environments, that are used to create and manage code.

[`http://www.economist.com/science/tq/displayStory.cfm?story_id=1841081`]

THE END

THE END .... really!