Theorem Proving for Verification

John Harrison
Intel Corporation

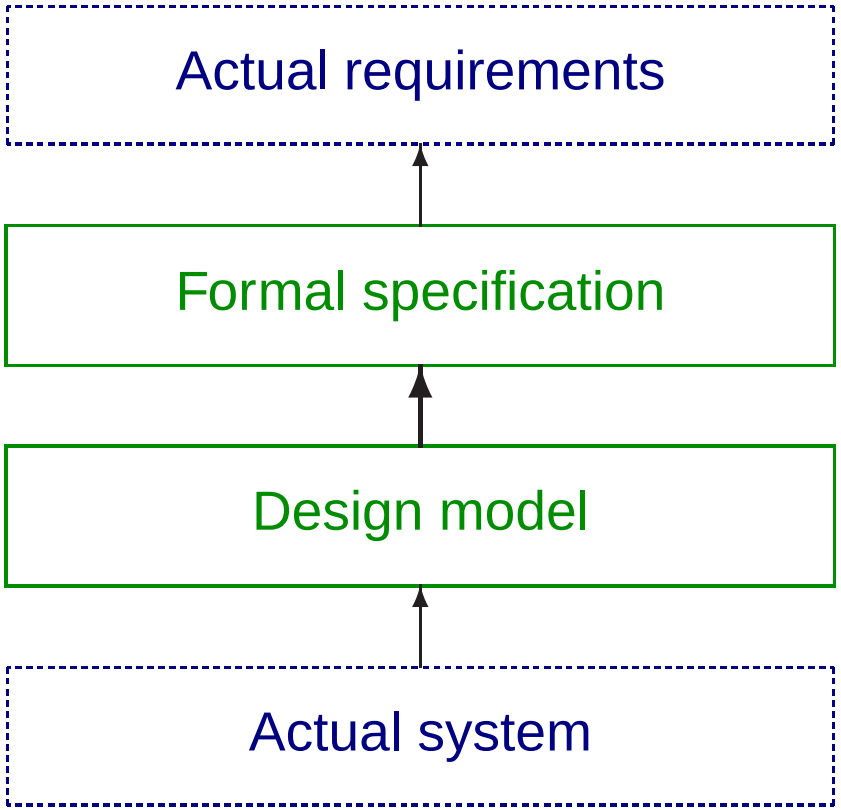
Galois talk (repeat of CAV tutorial)

Portland

16th September 2008 (10:30 – 12:00)

Formal verification

Formal verification: mathematically prove the correctness of a *design* with respect to a mathematical *formal specification*.



Essentials of formal verification

The basic steps in formal verification:

- Formally model the system
- Formalize the specification
- Prove that the model satisfies the spec

But what formalism should be used?

Some typical formalisms

- Propositional logic, a.k.a. Boolean algebra
- Temporal logic (CTL, LTL etc.)
- Quantifier-free combinations of first-order arithmetic theories
- Full first-order logic
- Higher-order logic or first-order logic with arithmetic or set theory

Expressiveness vs. automation

There is usually a roughly inverse relationship:

> The more expressive the formalism, the less the ‘proof’ is amenable to automation.

For the simplest formalisms, the proof can be so highly automated that we may not even think of it as ‘theorem proving’ at all.

The most expressive formalisms have a decision problem that is not decidable, or even semidecidable.

Logical syntax

English	Formal
false	$\bot$
true	$\top$
not p	$\neg p$
p and q	$p \wedge q$
p or q	$p \vee q$
p implies q	$p \Rightarrow q$
p iff q	$p \Leftrightarrow q$
for all x, p	$\forall x.\, p$
there exists x such that p	$\exists x.\, p$

Propositional logic

Formulas built up from *atomic propositions* (Boolean variables) and constants $\bot$, $\top$ using the propositional connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$.

No quantifiers or internal structure to the atomic propositions.

Propositional logic

Formulas built up from *atomic propositions* (Boolean variables) and constants $\bot$, $\top$ using the propositional connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$.

No quantifiers or internal structure to the atomic propositions.

A formula is a *tautology* if it is true for *all* assignments of truth values to the atomic propositions, e.g. $p \vee \neg p$ or $\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$.

A formula is *satisfiable* if it is true for *some* assignment of truth values to the atomic propositions.

Note that p is a tautology iff $\neg p$ is unsatisfiable.

Expressiveness of propositional logic

Propositional logic seems inexpressive but:

- Combinational logic circuits can be considered as Boolean formulas, and circuit equivalence as tautology/satisfiability testing
- Finite unfolding of sequential circuits or finite-state transition systems can be modelled using Boolean variables
- Many other arithmetical and combinatorial problems can be encoded in Boolean terms, e.g. primality testing, scheduling, planning.

The last item is a ‘practical’ counterpart to the theoretical significance of [co-]NP-complete problems.

Decision problem for propositional logic

Tautology/satisfiability checking is certainly decidable in exponential time, because we can examine all assignments of truth-values to the Boolean variables.

Unless $P = NP$, there is no polynomial-time decision procedure.

Algorithms like Davis-Putnam-Loveland-Logemann (DPLL) and Stålmarck's method are often surprisingly good in practice.

Embodied in highly tuned 'SAT solver' implementations, these have made a big impact, in formal verification and elsewhere.

First-order logic

Object-denoting *terms* built up from variables and constants denoting objects using function symbols, e.g. $x + 1$ or $f(x)$.

Atomic formulas are now built up by applying relations to terms, e.g. $x + 1 < 2 \cdot y$ or $R(f(x), g(y))$.

Can quantify over object-level variables, e.g. $\forall x.\, \exists y.\, \mathsf{loves}(x, y)$ or $\exists y.\, \forall x.\, \mathsf{loves}(x, y)$.

First-order logic

Object-denoting *terms* built up from variables and constants denoting objects using function symbols, e.g. $x + 1$ or $f(x)$.

Atomic formulas are now built up by applying relations to terms, e.g. $x + 1 < 2 \cdot y$ or $R(f(x), g(y))$.

Can quantify over object-level variables, e.g. $\forall x.\, \exists y.\, \mathsf{loves}(x, y)$ or $\exists y.\, \forall x.\, \mathsf{loves}(x, y)$.

The *first-order* means that we can't quantify over functions or relations, e.g. $\exists \mathsf{loves}.\, \forall x.\, \exists y.\, \mathsf{loves}(x, y)$.

A formula is *valid* when it holds for all *interpretations*, i.e. ways of interpreting the domain of objects as $D \neq \emptyset$, constants as elements of D, function symbols as functions $D^n \to D$ and relations as subsets of D^n, and *valuations* of the variables as elements of D.

First-order validity

There is no ‘naive’ algorithm for first-order validity, because we’d need to check all possible sets D, including infinite ones.

In fact, first-order validity is undecidable (Church/Turing).

First-order validity

There is no ‘naive’ algorithm for first-order validity, because we’d need to check all possible sets D, including infinite ones.

In fact, first-order validity is undecidable (Church/Turing).

On the other hand, it is *semidecidable* (r.e.), i.e. there are search algorithms that will in principle confirm that a valid formula is valid, but may run forever on invalid formulas.

- Tableaux
- Resolution

In practice, these can seldom solve ‘interesting’ problems in a practical time. Some notable successes such as McCune’s solution of the Robbins conjecture.

One interpretations versus all interpretations

We are often more interested in whether a formula holds in some *particular* interpretation or *particular class of* interpretations.

This is a very different problem, and it may be easier or harder than validity in all interpretations.

Consider first-order arithmetic formulas, using constants 0 and 1, function symbols $+$, $-$ and $\cdot$, and relation symbols $=$, $\leq$, $<$.

$\forall x.\, x = y \Rightarrow x = y$ holds in all interpretations

$\forall x.\, x + x = 2x$ holds in obvious arithmetic interpretations, but not all interpretations.

$\forall x.\, x \geq 0 \Rightarrow \exists y.\, x = y^2$ holds in $\mathbb{R}$ but not in $\mathbb{Z}$.

Different decision problems

Whether a first-order formula in the language of arithmetic:

- Holds in all interpretations: semidecidable (like first-order logic in general)

Different decision problems

Whether a first-order formula in the language of arithmetic:

- Holds in all interpretations: semidecidable (like first-order logic in general)
- Holds in $\mathbb{R}$: decidable (Tarski's quantifier elimination for real-closed fields)

Different decision problems

Whether a first-order formula in the language of arithmetic:

- Holds in all interpretations: semidecidable (like first-order logic in general)
- Holds in $\mathbb{R}$: decidable (Tarski's quantifier elimination for real-closed fields)
- Holds in all (ordered) rings: semidecidable (reduces to first-order validity; not decidable by interpretation)

Different decision problems

Whether a first-order formula in the language of arithmetic:

- Holds in all interpretations: semidecidable (like first-order logic in general)
- Holds in $\mathbb{R}$: decidable (Tarski's quantifier elimination for real-closed fields)
- Holds in all (ordered) rings: semidecidable (reduces to first-order validity; not decidable by interpretation)
- Holds in $\mathbb{Z}$: not even semidecidable (Gödel's theorem, or Tarski's theorem on the undecidability of truth).

Restricted decision problems

Some natural restrictions on undecidable problems can yield decidability:

- Although it's not even semidecidable if an arithmetic formula holds in $\mathbb{Z}$, it is decidable whether a purely *linear* formula does. (Formulas only involve multiplication by constants: Presburger arithmetic.)
- Although it's undecidable whether a formula holds in all rings, it is decidable for purely *universally quantified* formulas. (This is effectively the 'word problem' for rings and can be solved with a variant of Gröbner bases.)

These restricted fragments are often enough for practical problems.

Quantifier elimination

Some formulas involving quantifiers are equivalent to a quantifier-free one for some interpretation or class of interpretations:

- $(\exists x.\, ax^2 + bx + c = 0) \Leftrightarrow a \neq 0 \wedge b^2 \geq 4ac \vee a = 0 \wedge (b \neq 0 \vee c = 0)$ in $\mathbb{R}$
- $(\forall x.\, x < a \Rightarrow x < b) \Leftrightarrow a \leq b$ in all interpretations where $<$ is a dense total order.

Quantifier elimination

Some formulas involving quantifiers are equivalent to a quantifier-free one for some interpretation or class of interpretations:

- $(\exists x.\, ax^2 + bx + c = 0) \Leftrightarrow a \neq 0 \wedge b^2 \geq 4ac \vee a = 0 \wedge (b \neq 0 \vee c = 0)$ in $\mathbb{R}$
- $(\forall x.\, x < a \Rightarrow x < b) \Leftrightarrow a \leq b$ in all interpretations where $<$ is a dense total order.

For some classes of formulas we can find algorithmically for any formula a counterpart that is quantifier-free and equivalent in the class of interpretations: *quantifier elimination*.

For example, arithmetic formulas over $\mathbb{R}$, linear arithmetic formulas over $\mathbb{Z}$ (adding new 'divisibility by d' relations to the language).

Combining decision procedures

Even hitherto decidable fragments like Presburger arithmetic become undecidable if we combine with other function or relation symbols. With one unary function symbol we can characterize squaring;

$$(\forall n. f(-n) = f(n)) \wedge f(0) = 0 \wedge (\forall n. 0 \leq n \Rightarrow f(n+1) = f(n)+n+n+1)$$

and then multiplication by $m = n \cdot p \Leftrightarrow (n+p)^2 = n^2 + p^2 + 2m$

Combining decision procedures

Even hitherto decidable fragments like Presburger arithmetic become undecidable if we combine with other function or relation symbols. With one unary function symbol we can characterize squaring;

$$(\forall n. f(-n) = f(n)) \wedge f(0) = 0 \wedge (\forall n. 0 \leq n \Rightarrow f(n+1) = f(n)+n+n+1)$$

and then multiplication by $m = n \cdot p \Leftrightarrow (n+p)^2 = n^2 + p^2 + 2m$

However, for *universally quantified* ('quantifier-free') formulas, the *Nelson-Oppen* scheme lets us test validity in combinations of theories, e.g. arithmetic with uninterpreted functions.

$$u + 1 = v \wedge f(u) + 1 = u - 1 \wedge f(v - 1) - 1 = v + 1 \Rightarrow \bot$$

Implemented in SMT ('satisfiability modulo theories') solvers.

Higher-order logic

In *second-order logic* we allow quantification over functions and predicates with object-level arguments, e.g. in the induction principle

$$\forall P.\, P(0) \wedge (\forall n.\, P(n) \Rightarrow P(n+1)) \Rightarrow \forall n.\, P(n)$$

In *higher-order logic* we even allow predicates and functions with predicates and functions for arguments, and allow quantification over those, for infinitely many 'orders'.

Higher-order logic

In *second-order logic* we allow quantification over functions and predicates with object-level arguments, e.g. in the induction principle

$$\forall P.\, P(0) \wedge (\forall n.\, P(n) \Rightarrow P(n+1)) \Rightarrow \forall n.\, P(n)$$

In *higher-order logic* we even allow predicates and functions with predicates and functions for arguments, and allow quantification over those, for infinitely many 'orders'.

Since we can characterize the natural numbers up to isomorphism by second-order axioms, higher-order validity is not even semidecidable.

In practice, there are search procedures, essentially accepting some basic axioms for set theory. that will in principle find most 'ordinary' truths.

Higher-order logic as a universal formalism

HOL or set theory subsumes all the simpler formalisms like first-order logic and various arithmetic theories.

This even applies to temporal logics, which can be mapped directly into their ‘semantics’ in higher-order logic (*shallow embedding*).

Higher-order logic as a universal formalism

HOL or set theory subsumes all the simpler formalisms like first-order logic and various arithmetic theories.

This even applies to temporal logics, which can be mapped directly into their 'semantics' in higher-order logic (*shallow embedding*).

Map variables p of LTL to unary predicate variables p, which get applied to terms denoting 'times'.

- $(\Box\phi)(t) =_{def} \forall t'.\, t \leq t' \Rightarrow \phi(t')$
- $(\Diamond\phi)(t) =_{def} \exists t'.\, t \leq t' \wedge \phi(t')$
- $(\bigcirc\phi)(t) =_{def} \phi(t+1)$

Can state LTL validity by universally quantifying over all the predicates and constraining arithmetic operations to $\mathbb{N}$.

The state of automation

We have seen that for a rich formalism such as HOL, logical validity cannot *even in principle* be automated.

There are in principle search procedures that will confirm a wide class of interesting theorems. In practice these are seldom useful for practical problems.

Thus, for HOL and arguably even for FOL, we need *interactive* provers that can prove theorems with human guidance.

This means much more work for the human being.

At least we should try to make things easier by automating those bits of the proof that can be automated.

Interactive theorem proving

The idea of a more ‘interactive’ approach was already anticipated by pioneers, e.g. Wang (1960):

> [...] the writer believes that perhaps machines may more quickly become of practical use in mathematical research, not by proving new theorems, but by formalizing and checking outlines of proofs, say, from textbooks to detailed formalizations more rigorous that *Principia* [Mathematica], from technical papers to textbooks, or from abstracts to technical papers.

However, constructing an effective combination is not so easy. Early examples were the SAM series, AUTOMATH and Mizar.

Effective interactive theorem proving

What makes a good interactive theorem prover? Most agree on:

- Reliability
- Library of existing results
- Intuitive input format
- Powerful automated steps

Several other characteristics are more controversial:

- Programmability
- Checkability of proofs

LCF

One successful solution was pioneered in Edinburgh LCF ('Logic of Computable Functions').

The same 'LCF approach' has been used for many other theorem provers.

- Implement in a strongly-typed functional programming language (usually a variant of ML)
- Make `thm` ('theorem') an abstract data type with only simple primitive inference rules
- Make the implementation language available for arbitrary extensions.

Gives a good combination of extensibility and reliability.

Now used in Coq, HOL, Isabelle and several other systems.

Automated subsystems

Many of the leading interactive provers offer substantial automation of known decidable classes of subproblems.

HOL Light has basic but useful first-order proof search, and several decision methods, all built in a ‘correct by construction’ LCF way:

- Linear arithmetic over $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$
- Linear and nonlinear arithmetic over $\mathbb{C}$ and $\mathbb{R}$
- Universal formulas over integral domains and fields

Some interactive provers, notably PVS, have a full suite of combined decision procedures for quantifier-free theories.

It is possible to make use of external systems like computer algebra programs, soundly checking ‘certificates’ they produce.

The 17 Provers of the World

Freek Wiedijk's book *The Seventeen Provers of the World* (Springer-Verlag lecture notes in computer science volume 3600) describes:

HOL, Mizar, PVS, Coq, Otter/IVY, Isabelle/Isar, Alfa/Agda, ACL2, PhoX, IMPS, Metamath, Theorema, Lego, Nuprl, Omega, B prover, Minlog.

Each one has a proof that $\sqrt{2}$ is irrational.

There are many other systems besides these . . .

Why?

Even the best interactive theorem provers are **difficult to use**.

- For typical mathematics, far harder than writing an informal proof that a human mathematician would find acceptable.
- For most verifications, much more difficult and time-consuming than highly automated, almost ‘push-button’ methods like SAT, model checking, STE etc.

With this in mind, why make the effort?

Why?

Even the best interactive theorem provers are **difficult to use**.

- For typical mathematics, far harder than writing an informal proof that a human mathematician would find acceptable.
- For most verifications, much more difficult and time-consuming than highly automated, almost ‘push-button’ methods like SAT, model checking, STE etc.

With this in mind, why make the effort?

- Compared with an informal mathematical proof, they offer greater reliability and sometimes useful automation.
- Compared with highly automated verification methods, the use of a richer formalism can bring substantial benefits.

Benefits of theorem proving

- Richer formalism can express properties that are not, even in principle, in the scope of simpler formalisms or solvable by automated methods.

Benefits of theorem proving

- Richer formalism can express properties that are not, even in principle, in the scope of simpler formalisms or solvable by automated methods.
- We can formalize and verify properties including the underlying theory and assumptions, rather than isolated properties.

Benefits of theorem proving

- Richer formalism can express properties that are not, even in principle, in the scope of simpler formalisms or solvable by automated methods.
- We can formalize and verify properties including the underlying theory and assumptions, rather than isolated properties.
- It can be more efficient, since many supposedly automated methods require substantial time and human attention (BDD variable ordering etc.)

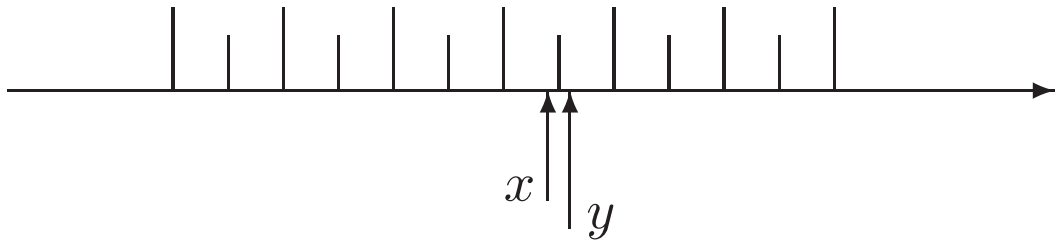
Benefits of theorem proving

- Richer formalism can express properties that are not, even in principle, in the scope of simpler formalisms or solvable by automated methods.
- We can formalize and verify properties including the underlying theory and assumptions, rather than isolated properties.
- It can be more efficient, since many supposedly automated methods require substantial time and human attention (BDD variable ordering etc.)
- It can be more intellectually stimulating since in the proof process one understands the design more deeply.

Avoiding limitations on expressiveness

Consider the definition of correctness of floating-point operations according to the IEEE Standard 754.

The key operation is rounding, and this is specified using forms of words like ‘the result is the closest representable number to the exact answer’.



Floating-point rounding

This specification is clear and intuitive, but *not executable*. We can only express it using real real numbers and sets.

For basic operations like addition, one can come up with executable variants expressible in more limited formalisms

Floating-point rounding

This specification is clear and intuitive, but *not executable*. We can only express it using real real numbers and sets.

For basic operations like addition, one can come up with executable variants expressible in more limited formalisms but:

- Almost collapses to a reference design, leaving one in doubt over correctness even of the spec.
- Impossible/infeasible for more complicated operations, even square root.

By using a general mathematical framework supporting general reasoning about real numbers, we can state the specification naturally and verify implementations with respect to it.

Avoiding limitations on automation

There are numerous model-checkers for verifying properties of finite-state transition systems.

These seldom directly support *infinite*-state transition systems. For example, Murphi does not even allow one to declare unbounded types (so one might consider this also as *expressiveness*).

Avoiding limitations on automation

There are numerous model-checkers for verifying properties of finite-state transition systems.

These seldom directly support *infinite*-state transition systems. For example, Murphi does not even allow one to declare unbounded types (so one might consider this also as *expressiveness*).

But in a general mathematical framework the key ideas are exactly the same in the finite-state and infinite state cases.

Of course, need new techniques of proof, e.g. inductive invariants, rigorous finite-state abstraction.

Verifying underlying theory

Verifications sometimes rest on non-trivial mathematics.

Without using a general theory, one has to deploy the required mathematical facts ‘by hand’, outside the verification, which is unsatisfactory and error-prone.

For example, floating-point arithmetic algorithms ofen depend on mathematical analysis and number theory:

- Accuracy of polynomial or rational approximations
- Analyzing hard-to-round cases, e.g. FP numbers close to multiples of $\pi/2$.

Also may need special-purpose automation.

Formalizing programming languages

Many traditional verification tools just assume some rules for program verification, e.g. Floyd-Hoare rules.

In a limited formalism, it's hard to justify why these hold for the language of interest.

By contrast, in a general framework one can specify the assumed semantics of the language and *prove* the Floyd-Hoare rules.

- *Shallow embedding* – map language directly to its semantics
- *Deep embedding* – formalize language syntax and semantics

Improving efficiency

Even if a system is verifiable *in principle* using automated finite-state techniques, it might not be practical *in practice*.

Consider a typical parametrized system with N essentially equivalent components (e.g. a cache coherence protocol with N similar cacheing agents).

Using finite-state techniques, we may in principle be able to perform exhaustive verifications for any particular N.

Improving efficiency

Even if a system is verifiable *in principle* using automated finite-state techniques, it might not be practical *in practice*.

Consider a typical parametrized system with N essentially equivalent components (e.g. a cache coherence protocol with N similar cacheing agents).

Using finite-state techniques, we may in principle be able to perform exhaustive verifications for any particular N.

However, for a complicated system, this may be practical only for moderate N, say $N \leq 3$.

The more analytic proofs encouraged by theorem proving may extend naturally to arbitrary N.

Improving understanding

Precisely because interactive theorem provers force the user to pay attention to all the details, one often ends up understanding the system more deeply.

For example, when verifying some FMA-based floating-point division algorithms based on Markstein's classic paper, we needed to formalize some of Markstein's theorems.

Improving understanding

Precisely because interactive theorem provers force the user to pay attention to all the details, one often ends up understanding the system more deeply.

For example, when verifying some FMA-based floating-point division algorithms based on Markstein's classic paper, we needed to formalize some of Markstein's theorems.

One theorem featured a hypothesis, but only a (well-known) consequence was used in the proof. By being forced to plug this gap, we noticed that the theorem could be more useful assuming only this consequence.

Improving understanding

Precisely because interactive theorem provers force the user to pay attention to all the details, one often ends up understanding the system more deeply.

For example, when verifying some FMA-based floating-point division algorithms based on Markstein's classic paper, we needed to formalize some of Markstein's theorems.

One theorem featured a hypothesis, but only a (well-known) consequence was used in the proof. By being forced to plug this gap, we noticed that the theorem could be more useful assuming only this consequence.

As a result we were able to design significantly more efficient algorithms that could be justified by the strengthened theorem.

Conclusions

We seem to face a clear trade-off between expressiveness of a logical formalism and the difficulty of its decision problem.

The most successful techniques in industry have been the highly automated applications of relatively inexpressive formalisms, perhaps hitting a ‘sweet spot’ with some temporal logics.

However, there is a substantial body of research on theorem-proving techniques. Some non-trivial problems still turn out to be decidable, but in general we need to settle for interactive human-guided proof.

The use of a general mathematical framework offers some significant advantages that can be significant, or even essential, for some verification tasks.