On Theorem Proving for Program Checking
Historical perspective and recent developments

Maria Paola Bonacina
Dipartimento di Informatica
Università degli Studi di Verona
Verona, Italy

July 28, 2010
Introduction

Where is theorem proving in program checking

Inside theorem proving
Decision procedures: Little engines of proof
Semi-decision procedures: Big engines of proof

Big and little engines together: a new theorem proving style

Decision procedures with speculative inferences

Current and future challenges
Program checking and theorem proving

- **Program checking**: Design computer programs that (help to) check whether computer programs satisfy desired properties.
Program checking and theorem proving

- **Program checking**: Design computer programs that (help to) check whether computer programs satisfy desired properties

- **Theorem proving**: Design computer programs that (help to) check whether formulæ follow from other formulæ
Some motivation for program checking

- Software is everywhere
- Needed: *Reliability*
- Difficult goal: Software may be
  - Artful
  - Complex
  - Huge
  - Varied
  - Old (and undocumented)
  - Less standardized than hardware
Historical roots: program checking

Historical roots: program checking

Historical roots: theorem proving

Historical roots: theorem proving

After four decades of research ...

Many approaches to program checking:

- **Testing**: automated test case generation, (semi-)automated testing ...
After four decades of research ...

Many approaches to program checking:

- **Testing**: automated test case generation, (semi-)automated testing ...

- **Static analysis**: type systems, data-flow analysis, control-flow analysis, pointer analysis, symbolic execution, abstract interpretation ...
After four decades of research ...

Many approaches to program checking:

- **Testing**: automated test case generation, (semi-)automated testing ...
- **Static analysis**: type systems, data-flow analysis, control-flow analysis, pointer analysis, symbolic execution, abstract interpretation ...
- **Dynamic analysis**: traces, abstract interpretation ...
After four decades of research ... 

Many approaches to program checking:

- *Testing*: automated test case generation, (semi-)automated testing ...
- *Static analysis*: type systems, data-flow analysis, control-flow analysis, pointer analysis, symbolic execution, abstract interpretation ...
- *Dynamic analysis*: traces, abstract interpretation ...
- *Software model checking*: BMC, CEGAR, SMT-MC ...
After four decades of research ...

Many approaches to program checking:

- **Testing**: automated test case generation, (semi-)automated testing ...

- **Static analysis**: type systems, data-flow analysis, control-flow analysis, pointer analysis, symbolic execution, abstract interpretation ...

- **Dynamic analysis**: traces, abstract interpretation ...

- **Software model checking**: BMC, CEGAR, SMT-MC ...

- **Deductive verification**: weakest precondition calculi, verification conditions generation and proof ...
First summary

- A pipeline of tools for program checking, where
  - Problems of increasing difficulty are attacked by
  - Approaches of increasing power (and cost)
First summary

- A *pipeline of tools* for program checking, where
  - Problems of increasing difficulty are attacked by
  - Approaches of increasing power (and cost)
- Most methods for program checking apply logic
First summary

- A *pipeline of tools* for program checking, where
  - Problems of increasing difficulty are attacked by
    - Approaches of increasing power (and cost)
  - Most methods for program checking apply logic
  - Most can benefit from theorem proving
First summary

- A *pipeline of tools* for program checking, where
  - Problems of increasing difficulty are attacked by
    - Approaches of increasing power (and cost)
- Most methods for program checking apply logic
- Most can benefit from theorem proving
- Theorem proving *is* artificial intelligence
- Theorem proving for program checking *is* artificial intelligence
Program checking and theorem proving
Software model checking with predicate abstraction

- Original model checking: finite state machine
- Software: infinitely many states
- How to finitize? Abstraction
- Model check *abstract program*
- Abstract counter-example + formula $\varphi$ sat iff also concrete counter-example
- Apply theorem prover: if $\varphi$ unsat refine abstraction with predicates from proof
More theorem proving in model checking

- No abstraction: finite representation by formulæ with quantifiers
- Backward reachability: from set of error states towards initial states
- Does pre-image of error states intersect with set of initial state?
- Did the computation of the pre-image reach a fixed point?
- Reduced to satisfiability of formulæ with quantifiers
Deductive verification

- The program is annotated with *assertions*
- Program variables appear in assertions as *free variables* (constants in refutational theorem proving)
- Program *state*: an assignment to free variables, hence an *interpretation*
Verifying compiler + theorem prover

- Given: annotated program
- Decomposition into basic paths
- Backward propagation by computing weakest pre-conditions
- Verification condition: the given pre-condition implies the computed one
- If the verification conditions are valid, the annotations are *invariants*
- Otherwise, counter-model is useful to find error in program or annotations
From invariant checking to invariant generation

- Manual annotation of programs is tedious and expensive
- Programmers may appreciate writing functional specifications, not loop invariants, run-time assertions, function call assertions
- Automated annotation
- Automated generation of valid annotations, that is, *invariants*
Given: partially annotated program
- Decomposition into basic paths
- Forward propagation by computing strongest post-conditions
- Does the computed post-condition imply the given one?
- Answer by theorem proving
- If not, update the post-condition
Abstract interpretation

- Trade-off between precision and termination: abstraction
- Abstract interpretation: restrict language of admissible formulæ to an *abstract domain* (syntactically restricted class of formulæ)
Second summary

- There is much theorem proving in SW model checking
- Program checking use theorem prover as back-end reasoner
- Theorem prover must be decision procedure
- Model building as important as proof building
- Abstraction as a way to make satisfiability decidable
- However, problems may contain quantifiers: tension between expressivity and decidability
Inside theorem proving
Decision procedures

- Davis-Putnam-Logemann-Loveland (DPLL) procedure for SAT
- $\mathcal{T}$-solver: Satisfiability procedure for $\mathcal{T}$
  Equality: congruence closure (CC)
- DPLL($\mathcal{T}$)-based SMT-solver: Decision procedure for $\mathcal{T} = \bigcup_{i=1}^{n} \mathcal{T}_i$ with
- Nelson-Oppen combination of $\mathcal{T}_i$-sat procedures
Propositional logic

- Build candidate model $M$
- Decision procedure:
  - model found: return $sat$
  - failure: return $unsat$
- Depth-first search with backtracking
DPLL

State of derivation: $M \parallel F$

- **Decide**: guess $L$ is true, add it to $M$ (decided literal)
- **UnitPropagate**: propagate consequences of assignment (implied literals)
- **Conflict**: detect $L_1 \lor \ldots \lor L_n$ all false
- **Explain**: unfold implied literals in conflict clause by resolution
- **Learn** conflict clause $C \lor L$
- **Backjump**: when only $L$ assigned at current decision level, jump back to least recent level where $C$ false and $L$ unassigned, undo at least one decision, make $L$ true (implied by $C \lor L$)
- **Unsat**: conflict clause is $\Box$ (nothing else to try)
DPLL($\mathcal{T}$)

State of derivation: $M \parallel F$

- **$\mathcal{T}$-Propagate**: add to $M$ an $L$ that is $\mathcal{T}$-consequence of $M$
- **$\mathcal{T}$-Conflict**: detect that $L_1, \ldots, L_n$ in $M$ are $\mathcal{T}$-inconsistent
Equality sharing method (Nelson-Oppen)

- $T_i$’s disjoint: no shared function/predicate symbols beside $\simeq$
- Mixed terms separated by introducing new constants
- $T_i$-solvers generate and propagate all entailed (disjunctions of) equalities between shared constants
- $T_i$’s stably infinite: every $T_i$-sat ground formula has $T_i$-model with infinite cardinality (ensures existence of quantifier-free interpolants hence that propagation suffices in completeness proof)
Model-based theory combination

A variant of equality sharing (rule PropagateEq):

- Generating (disjunctions of) equalities true in all $\mathcal{T}_i$-models consistent with $M$ may be expensive
- If each $\mathcal{T}_i$-solver builds a candidate $\mathcal{T}_i$-model $M_i$
- Generate and propagate equalities true in $M_i$
- Optimistic: if equality turns out to be inconsistent, backtrack

[Leonardo de Moura and Nikolaj Bjørner 2007]
Third summary

- SMT-solvers are theorem provers
- They do model building: both DPLL and CC
- Model-driven or context-driven deduction and simplification
- Especially good at theories such as linear arithmetic and bit-vectors, and integrating them with SAT
- Conceived for SAT and ground problems, not for quantifiers
Superposition-based inference system $\Gamma$

- Generic, FOL$^+=\$, axiomatized theories
- Deduce clauses from clauses (expansion)
- Remove redundant clauses (contraction)
- Well-founded ordering $\succ$ on terms and literals to restrict expansion and define contraction
- Semi-decision procedure
- No backtracking
Inference system $\Gamma$

State of derivation: set of clauses $F$

- Resolution
- Superposition/Paramodulation: resolution with equality built-in
- Simplification: by well-founded rewriting
- Subsumption: eliminate more general clauses
- Other rules: e.g., Factoring rules, Deletion of trivial clauses
Big engines as little engines

- *Termination* results by analysis of inferences: $\Gamma$ is $\mathcal{T}$-satisfiability procedure

- Covered theories include: lists, arrays and records with or without extensionality, recursive data structures

Joint works with Alessandro Armando, Mnacho Echenim, Michaël Rusinowitch, Silvio Ranise and Stephan Schulz
Also for combination of theories

- **Theorem (Modularity of termination):** if $\Gamma$ terminates on $\mathcal{R}_i$-sat problems, it terminates also on $\mathcal{R}$-sat problems for $\mathcal{R} = \bigcup_{i=1}^{n} \mathcal{R}_i$, if the $\mathcal{R}_i$'s are disjoint and variable-inactive.
- Variable-inactivity: no maximal literals of the form $t \simeq x$ where $x \notin Var(t)$ (no paramodulation from variables).
- The only inferences across theories are *superpositions from shared constants* (correspond to equalities between shared constants in equality sharing).

Joint work with Alessandro Armando, Silvio Ranise and Stephan Schulz
Variable inactivity implies stable infiniteness

- **Theorem:** if $\mathcal{R}$ is variable-inactive, then it is stably infinite
- $\Gamma$ reveals lack of stable infiniteness by generating a *cardinality constraint* (e.g., $y \simeq x \lor y \simeq z$) which is not variable-inactive

Joint work with Silvio Ghilardi, Enrica Nicolini, Daniele Zucchelli 2006
Fourth summary

- Resolution/superposition-based engines good for reasoning on formulæ with quantified variables: *automated* instantiation
- Not for large non-Horn clauses
- Not for theories such as linear arithmetic or bit-vectors
- Unexpected: they are satisfiability-procedures for theories such as lists, arrays, records and their combinations
Big and little engines together: a new theorem proving style
Problem statement

- Decide *satisfiability* of first-order formulæ generated by *verifying compilers* or *static analyzer*
- Satisfiability w.r.t. *background theories*
- With *quantifiers* to write, e.g.,
  - invariants about loops, heaps, data structures ...  
  - axioms of *type systems* or *application-specific theories* without decision procedure
- Emphasis on *automation*: prover called by other tools
Typical verification problem

- Background theory $\mathcal{T}$
  - $\mathcal{T} = \bigcup_{i=1}^{n} \mathcal{T}_i$, e.g., linear arithmetic
- Set of formulæ: $\mathcal{R} \cup \mathcal{P}$
  - $\mathcal{R}$: set of non-ground clauses without $\mathcal{T}$-symbols
  - $\mathcal{P}$: large ground formula (set of ground clauses) with $\mathcal{T}$-symbols
- Determine whether $\mathcal{R} \cup \mathcal{P}$ is satisfiable modulo $\mathcal{T}$
  (Equivalently: determine whether $\mathcal{T} \cup \mathcal{R} \cup \mathcal{P}$ is satisfiable)
A new theorem proving style

- Given the kind of problem
- Given the complementary strengths of SMT-solvers and resolution/superposition based theorem provers
- Put them together!
- A few approaches
  - DPLL(Γ + T)
  - LASCA ([Konstatin Korovin and Andrei Voronkov 2007-09]), SUP(LA) ([Christoph Weidenbach et al. 2009]) ...
DPLL(Γ+T): integrate Γ in DPLL(T)

- **Idea:** literals in $M$ can be premises of Γ-inferences
- Stored as *hypotheses* in inferred clause
- **Hypothetical clause:** $(L_1 \land \ldots \land L_n) \triangleright (L'_1 \lor \ldots \lor L'_m)$
  interpreted as $\neg L_1 \lor \ldots \lor \neg L_n \lor L'_1 \lor \ldots \lor L'_m$
- Inferred clauses inherit hypotheses from premises

Joint work with Leonardo de Moura and Chris Lynch
building on top of work by Nikolaj Bjørner and Leonardo de Moura
State of derivation: $M \parallel F$

- **Expansion**: take as premises non-ground clauses from $F$ and $\mathcal{R}$-literals (unit clauses) from $M$ and add result to $F$
- **Backjump**: remove hypothetical clauses depending on undone assignments
- **Contraction**: as above + scope level to prevent situation where clause is deleted, but clauses that make it redundant are gone because of backjumping
Completeness of DPLL(\(\Gamma + \mathcal{T}\))

- **Refutational completeness** of the inference system:
  - from that of \(\Gamma\), DPLL(\(\mathcal{T}\)) and equality sharing
  - made combinable by variable-inactivity

- **Fairness** of the search plan:
  - depth-first search fair only for ground SMT problems;
  - add *iterative deepening* on *inference depth*
Fifth summary

Use each engine for what is best at:

- $\text{DPLL}(T)$ works on ground clauses
- $\Gamma$ not involved with ground inferences and built-in theories
- $\Gamma$ works on non-ground clauses and ground unit clauses taken from $M$: also $\Gamma$-inferences are context-driven
- $\Gamma$ works on $\mathcal{R}$-sat problem
- Completeness: showed how to integrate Nelson-Oppen built-in theories and variable-inactive axiomatized theories
Decision procedures with speculative inferences
Problematic axioms do occur in relevant inputs

Example:

1. $\neg(x \sqsubseteq y) \lor f(x) \sqsubseteq f(y)$ (*Monotonicity*)
2. $a \sqsubseteq b$ generates by resolution
3. $\{f^i(a) \sqsubseteq f^i(b)\}_{i \geq 0}$

E.g. $f(a) \sqsubseteq f(b)$ or $f^2(a) \sqsubseteq f^2(b)$ often suffice to show satisfiability
Idea: Allow speculative inferences

1. \( \neg (x \sqsubseteq y) \lor f(x) \sqsubseteq f(y) \)
2. \( a \sqsubseteq b \)
3. \( a \sqsubseteq f(c) \)
4. \( \neg (a \sqsubseteq c) \)
Idea: Allow speculative inferences

1. \( \neg (x \sqsubseteq y) \lor f(x) \sqsubseteq f(y) \)
2. \( a \sqsubseteq b \)
3. \( a \sqsubseteq f(c) \)
4. \( \neg (a \sqsubseteq c) \)

1. Add \( f(x) \simeq x \)
2. Rewrite \( a \sqsubseteq f(c) \) into \( a \sqsubseteq c \) and get \( \square \): backtrack!
Idea: Allow speculative inferences

1. $\neg(x \sqsubseteq y) \lor f(x) \sqsubseteq f(y)$
2. $a \sqsubseteq b$
3. $a \sqsubseteq f(c)$
4. $\neg(a \sqsubseteq c)$

1. Add $f(x) \simeq x$
2. Rewrite $a \sqsubseteq f(c)$ into $a \sqsubseteq c$ and get $\Box$: backtrack!
3. Add $f(f(x)) \simeq x$
4. $a \sqsubseteq b$ yields only $f(a) \sqsubseteq f(b)$
5. $a \sqsubseteq f(c)$ yields only $f(a) \sqsubseteq c$
6. Terminate and detect satisfiability
Speculative inferences in DPLL($\Gamma + \mathcal{T}$)

- Speculative inference to induce termination on sat input
- What if it makes problem unsat?!
- Detect conflict and backjump:
  - Keep track by adding $[C] \triangleright C$
  - $[C]$: new propositional variable (a “name” for $C$)
  - Speculative inferences are reversible
- Rule *SpeculativeIntro* also bounded by iterative deepening
Example as done by system

1. $\neg(x \sqsubseteq y) \lor f(x) \sqsubseteq f(y)$
2. $a \sqsubseteq b$
3. $a \sqsubseteq f(c)$
4. $\neg(a \sqsubseteq c)$
Example as done by system

1. \( \neg(x \sqsubseteq y) \lor f(x) \sqsubseteq f(y) \)
2. \( a \sqsubseteq b \)
3. \( a \sqsubseteq f(c) \)
4. \( \neg(a \sqsubseteq c) \)

1. Add \( [f(x) \simeq x] \triangleright f(x) \simeq x \)
2. Rewrite \( a \sqsubseteq f(c) \) into \( [f(x) \simeq x] \triangleright a \sqsubseteq c \)
Example as done by system

1. \( \neg(x \sqsubseteq y) \lor f(x) \sqsubseteq f(y) \)
2. \( a \sqsubseteq b \)
3. \( a \sqsubseteq f(c) \)
4. \( \neg(a \sqsubseteq c) \)

1. Add \( \llbracket f(x) \simeq x \rrbracket \triangleright f(x) \simeq x \)
2. Rewrite \( a \sqsubseteq f(c) \) into \( \llbracket f(x) \simeq x \rrbracket \triangleright a \sqsubseteq c \)
3. Generate \( \llbracket f(x) \simeq x \rrbracket \triangleright \Box\); Backtrack, learn \( \neg\llbracket f(x) \simeq x \rrbracket \)
Example as done by system

1. \( \neg(x \sqsubseteq y) \lor f(x) \sqsubseteq f(y) \)
2. \( a \sqsubseteq b \)
3. \( a \sqsubseteq f(c) \)
4. \( \neg(a \sqsubseteq c) \)

1. Add \( [f(x) \simeq x] \triangleright f(x) \simeq x \)
2. Rewrite \( a \sqsubseteq f(c) \) into \( [f(x) \simeq x] \triangleright a \sqsubseteq c \)
3. Generate \( [f(x) \simeq x] \triangleright \Box \); Backtrack, learn \( \neg[f(x) \simeq x] \)
4. Add \( [f(f(x)) \simeq x] \triangleright f(f(x)) \simeq x \)
5. \( a \sqsubseteq b \) yields only \( f(a) \sqsubseteq f(b) \)
6. \( a \sqsubseteq f(c) \) yields only \( f(a) \sqsubseteq f(f(c)) \)
    rewritten to \( [f(f(x)) = x] \triangleright f(a) \sqsubseteq c \)
7. Terminate and detect satisfiability
How to get decision procedures

To decide satisfiability modulo $T$ of $R \cup P$:

- Find sequence of “speculative axioms” $U$
- Show that there exists $k$ s.t. $k$-bounded DPLL($\Gamma+T$) is guaranteed to terminate
  - with $Unsat$ if $R \cup P$ is $T$-unsat
  - in a state which is not stuck at $k$ if $R \cup P$ is $T$-sat
Axiomatizations of type systems

- **Reflexivity** \( x \sqsubseteq x \) (1)
- **Transitivity** \( \neg (x \sqsubseteq y) \lor \neg (y \sqsubseteq z) \lor x \sqsubseteq z \) (2)
- **Anti-Symmetry** \( \neg (x \sqsubseteq y) \lor \neg (y \sqsubseteq x) \lor x \simeq y \) (3)
- **Monotonicity** \( \neg (x \sqsubseteq y) \lor f(x) \sqsubseteq f(y) \) (4)
- **Tree-Property** \( \neg (z \sqsubseteq x) \lor \neg (z \sqsubseteq y) \lor x \sqsubseteq y \lor y \sqsubseteq x \) (5)

**Multiple inheritance:** \( \text{MI} = \{(1), (2), (3), (4)\} \)

**Single inheritance:** \( \text{SI} = \text{MI} \cup \{(5)\} \)
Concrete examples of decision procedures

DPLL(Γ+T) with \textit{SpeculativIntro} adding $f^j(x) \simeq f^k(x)$ for $j > k$ decides the satisfiability modulo $T$ of problems

- MI $\cup$ P
- SI $\cup$ P

Joint work with Leonardo de Moura and Chris Lynch
Current and future challenges in program checking

- Improve *expressivity*, *scalability*, *precision* and *automation*
Current and future challenges in program checking

- Improve expressivity, scalability, precision and automation
- Integration of model checking and theorem proving
- Integration of abstract interpretation and theorem proving
Current and future challenges in program checking

- Improve expressivity, scalability, precision and automation
- Integration of model checking and theorem proving
- Integration of abstract interpretation and theorem proving
- Cooperation of verification and synthesis
- Software/hardware border: blurred, evolving
Current and future challenges in theorem proving

- For DPLL($\Gamma + \mathcal{T}$):
  - A top-notch implementation
  - More decision procedures
Current and future challenges in theorem proving

- For DPLL($\Gamma + \mathcal{T}$):
  - A top-notch implementation
  - More decision procedures
- Automation and interaction
Current and future challenges in theorem proving

- For DPLL($\Gamma + \mathcal{T}$):
  - A top-notch implementation
  - More decision procedures
- Automation and interaction
- Embedded theorem proving
Acknowledgements

Thanks to my co-authors

and

Thank you!