Inference Architectures for Satisfiability Modulo Theories

N. Shankar (with Bruno Dutertre)

Computer Science Laboratory
SRI International
Menlo Park, CA

July 20, 2010

Robin and Amir

Overview

- Designing software architectures is more art than science.
- The challenges are especially acute for inference architectures.
- Inference components can interact in ways that bend and break modularity.
- This talk covers inference architectures both at the micro-level and macro-level.
- At the micro-level, we focus on architectures for satisfiability modulo theories (SMT).
- At the macro-level, we describe two ongoing projects
  1. The Evidential Tool Bus (ETB) for coarse-grained integration of inference components
  2. The Kernel of Truth (KoT) for certifying claims from inference tools

SMT and Architecture

- SMT solvers are extremely useful for a number of applications.
- Such solvers are not easy to build/maintain.
- We (SRI) have been involved in building SMT solvers for a very long time.
- And we're still trying to get the architecture right.
- Existing architecture (e.g., Yices 1 and 2) for SMT solvers have important limitations.
- We outline a new architecture for SMT that will serve as the basis for the next version of Yices.
- The ideas in this talk are inspired by various SMT implementations over the years, from Shostak's STP to Yices 1 and 2.

Should We Care About Architecture?

- Any complex piece of software lives and dies by the quality of its design.
- The design elements include the API, architecture, algorithms, and data structures.
- In the last few years, there has been good progress on all of these fronts.
- But current SMT solvers are not even adequate to replace the 30-year-old Shostak implementation currently used in PVS.
- In these solvers, the theory solver is subordinate to the SAT solver.
- We outline a new framework that is centered around the *e-graph module* that maintains equality/disequality information.

Historical Perspective

- Shostak (1979) introduced the idea of a combined decision procedure with equality and arithmetic.
- Nelson and Oppen (1979) introduced a technique for combining disjoint theory decision procedures
  1. Input constraint ϕ is purified as $\phi_1 \wedge \ldots \wedge \phi_n$ where each ϕ_i is a constraint in theory i.
  2. If for some arrangement A of equalities on the shared variables, each $A \wedge \phi_i$ is satisfiable in theory i, then ϕ is satisfiable in the union of the theories.
- Shostak (1982/84; corrected/explained in Rueß/S 2001/02) introduced a way of combining disjoint theory decision procedures constructed using solvers and canonizers.
- Solvers reduce equalities to solved form, and canonizers simplify arbitrary terms to normal form *modulo* the solution state (context).

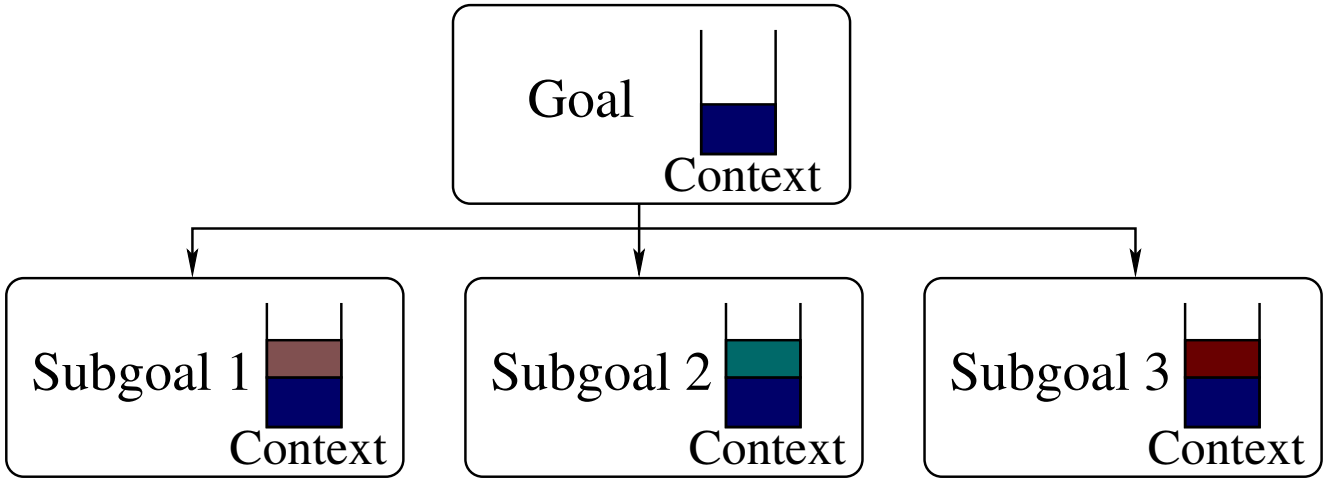
SAT + Theory Solvers

- SMT solvers in the Stanford Pascal Verifier (1979) and Simplify (mid-90s) used the Nelson/Oppen method.
- Shostak's method was part of the SMT solver in the (original) STP (1982) solver, and later in the EHDM (1983-88) and ICS (2002) SMT solvers.
- Both methods used an *e-graph* structure to maintain and propagate equality information.
- Wolfman and Weld's LPSAT (1999) was the first serious DPLL-based SMT solver, and the noughties saw a flurry of new ones (Verifun, ICS, MathSAT, CVC, ...).

Shostak in PVS

- Shostak's architecture was convenient for use in interactive proof.
- The context consists of a *usealist*, *findalist*, and *sigalist*, and these were implemented as stacks.
- Terms and formulas can be *canonized* relative to the context,
- Literals can be asserted to a context to yield a new context without affecting the old one.
- Push/Pop are easily implemented since retraction is just popping the stack.
- Shostak's implementation was also more open-ended and incremental than Nelson/Oppen implementations.

Shostak in PVS



- In PVS, each sequent has a decision procedure context, but the stack architecture enables safe context-sharing.
- The canonizer is also used heavily within PVS to test for contextual equality.
- PVS has a contextual rewriter/simplifier that exploits the conditional and predicate subtype information available in a context.

Fast Forward

- Yices is many orders of magnitude faster, and significantly more robust.
- But it still lacks some of the functionality of Shostak.
- PVS can invoke Yices as an oracle, but only to terminate a proof goal, not to simplify it.
- Some of the Shostak functionality can be recovered with a tighter embedding of Yices in PVS.
- But a different architecture would benefit both Yices and its clients.

Language

- Signature $\Sigma[X]$ contains functions and predicate symbols with associated arities, and X is a set of variables.
- The signature can be used to construct
  - *Terms* $\tau := x \mid f(\tau_1, \ldots, \tau_n)$
  - *Atoms* $\alpha := p(\tau_1, \ldots \tau_n)$,
  - *Literals* $\lambda := \alpha \mid \neg\alpha$
  - *Constraints* $\lambda_1 \wedge \ldots \wedge \lambda_n$,
  - *Clauses* $\lambda_1 \vee \ldots \vee \lambda_n$,
  - *Formulas* $\psi := p(\tau_1, \ldots, \tau_n) \mid \tau_0 = \tau_1 \mid \neg\psi_0 \mid \psi_0 \vee \psi_1 \mid \psi_0 \wedge \psi_1 \mid (\exists x : \psi_0) \mid (\forall x : \psi_0)$

Semantics

- A Σ-structure M consists of
  - A *domain* $|M|$
  - A map $M(f)$ from $|M|^n \rightarrow M$ for each n-ary function $f \in \Sigma$
  - A map $M(p)$ from $|M|^n \rightarrow \{\top, \bot\}$ for each n-ary predicate p.

  $\Sigma[X]$-structure M also maps variables in X to domain elements in $|M|$.
- The interpretation of terms and formulas in M is standard.
- With this, we have $M \models \psi$ when M satisfies formula ψ.
- A theory τ has a signature Σ_τ and a class of models $\mathcal{M}_\tau$.

Inference System

- An inference system $\mathcal{I}$ for a Σ-theory $\mathcal{T}$ is a $\Sigma[X]$-inference structure $\langle \Psi, \Lambda, \vdash \rangle$ that is
  1. **Conservative:** Whenever $\varphi \vdash_{\mathcal{I}} \varphi'$, $\Lambda(\varphi)$ and $\Lambda(\varphi')$ are $\mathcal{T}$-equisatisfiable.
  2. **Progressive:** The reduction relation $\vdash_{\mathcal{I}}$ should be well-founded, i.e., infinite sequences of the form $\langle \varphi_0 \vdash \varphi_1 \vdash \varphi_2 \vdash \ldots \rangle$ must not exist.
  3. **Canonizing:** A state is irreducible only if it is either $\bot$ or is $\mathcal{T}$-satisfiable.
- *For any class of $\Sigma[X]$-formulas Ψ, if there is a mapping ν from Ψ to Φ such that $\Lambda(\nu(A)) = A$, then a $\mathcal{T}$-inference system is a sound and complete decision procedure for $\mathcal{T}$-satisfiability in Ψ (given a function f such that $\kappa \vdash f(\kappa)$ when there is a κ' such that $\kappa \vdash \kappa'$) .*

Conflict-Driven Clause Learning (CDCL) SAT

Name	Rule	Condition
Propagate	$\dfrac{h, \langle M \rangle, K, C}{h, \langle M, l[\Gamma] \rangle, K, C}$	$\Gamma \equiv l \vee \Gamma' \in K \cup C$ $M \models \neg\Gamma'$
Decide	$\dfrac{h, \langle M \rangle, K, C}{h+1, \langle M; l[] \rangle, K, C}$	$M \not\models l$ $M \not\models \neg l$
Conflict	$\dfrac{0, \langle M \rangle, K, C}{\bot}$	$M \models \neg\Gamma$ for some $\Gamma \in K \cup C$
Backjump	$\dfrac{h+1, \langle M \rangle, K, C}{h', \langle M_{\leq h'}, l[\Gamma'] \rangle, K, C \cup \{\Gamma'\}}$	$M \models \neg\Gamma$ for some $\Gamma \in K \cup C$ $\langle h', \Gamma' \rangle$ $= analyze(\psi)(\Gamma)$ for $\psi = h, \langle M \rangle, K, C$

Example Inference Systems

- Inference systems help structure the correctness arguments.
- Several theoretical results are in *Modularity and refinement in inference systems* [Ganzinger, R, S].
- Simplifiers are inference systems without canonicity.
- Many inference algorithms can be described as inference systems, e.g.,
  1. Union-find for equality
  2. Propositional resolution
  3. Basic superposition for equality/propositional reasoning
  4. CDCL
  5. Simplex-based linear arithmetic reasoning
  6. SMT

SMT Overview

- In SMT solving, the Boolean atoms represent constraints over individual variables ranging over integers, reals, datatypes, and arrays.
- The constraints can involve theory operations, equality, and inequality.
- The SAT solver has to interact with a theory constraint solver which propagates truth assignments and adds new clauses.
- The theory solver can detect conflicts involving theory reasoning, e.g.,
  1. $f(x) = f(y) \vee x \neq y$
  2. $f(x-2) \neq f(y+3) \vee x - y \leq 5 \vee y - z \leq -2 \vee z - x \leq -3$
  3. $x \; XOR \; y \neq 0b0000000 \vee select(store(A, x, v), y) = v$
- The theory solver must produce efficient explanations, incremental assertions, and efficient backtracking.

Example Constraint Solvers

- **Core theory:** Equalities between variables $x = y$, offset equalities $x = y + c$.
- **Term equality:** Congruence closure for uninterpreted function symbols
- **Difference constraints:** Incremental negative cycle detection for inequality constraints of the form $x - y \leq k$.
- **Linear arithmetic constraints:** Fourier's method, Simplex.
- **Bit Vectors:** Bit-blasting

Theory Constraint Solver Interface

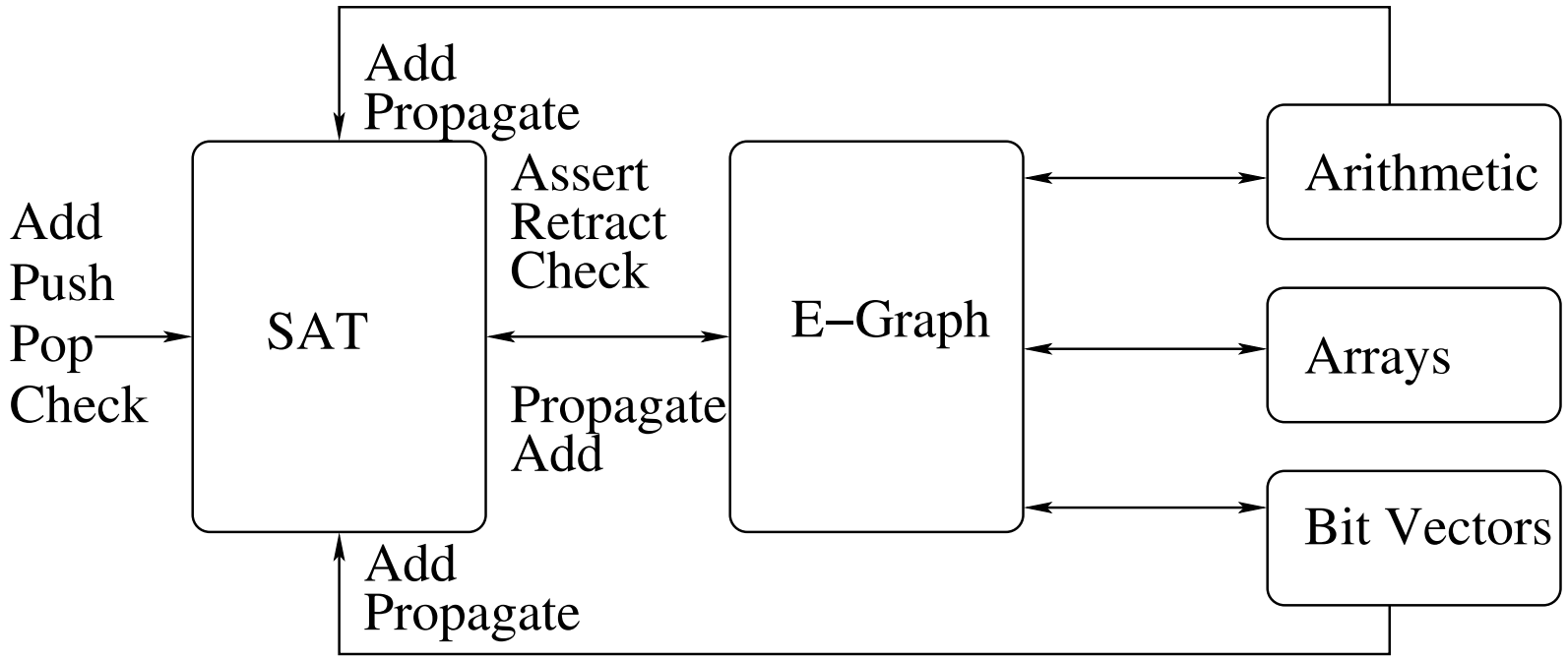
The satisfiability procedure uses a theory constraint solver oracle which maintains the theory state S with the interface operations:

1. $\mathit{assert}(l, S)$ adds literal l to the theory state S returning a new state S' or $\bot[\Delta]$
2. $\mathit{check}(S)$ checks if the conjunction of literals asserted to S is satisfiable, and returns either $\top$ or $\bot[\Delta]$.
3. $\mathit{retract}(S, l)$: Retracts, in reverse chronological order, the assertions up to and including l from state S.
4. $\mathit{model}(S)$: Builds a model for a state known to be satisfiable.

Satisfiability Modulo Theories

- SMT deals with formulas with theory atoms like $x = y$, $x \neq y$, $x - y \leq 3$, and $\mathit{select}(\mathit{store}(A, i, v), j) = w$.
- The DPLL search state is augmented with a *theory state* S in addition to the partial assignment.
- Total assignments are *checked* for theory satisfiability.
- When a literal is added to M by unit propagation, it is also *asserted* to S.
- When a literal is implied by S, it is *propagated* to M.
- When backjumping, the literals deleted from M are also *retracted* from S.

SMT Architecture (Yices 2)



Preprocessing is the dirty little secret of SMT solving.
In some categories, 30% of the problems are directly seen to be unsatisfiable by simplification.

The E-Graph Structure (Yices 2)

- Given a set of equalities $f(select(store(A, i, v), j)) = w, f(u) = w - 2, i = j, u = v$
- The e-graph contains the terms $x, y, z, k, f(x), w, f(u), y, z, i, j, u, v$.
- The bindings $x \mapsto select(store(A, i, v), j)$, $y \mapsto w - 2$, and the equivalences $z \sim f(x)$ and $k \sim f(u)$ are maintained by the e-graph.
- The e-graph is a congruence-closed graph with $f(x) \sim w, f(u) \sim y, i \sim j, u \sim v$.
- The array theory infers that $select(store(A, i, v), j) = v$.
- The arithmetic theory infers that $k \neq w$.
- The discrepancy between arithmetic and e-graph models leads to a split $u = v$ yielding an arithmetic conflict in one branch, and an e-graph conflict in the other.

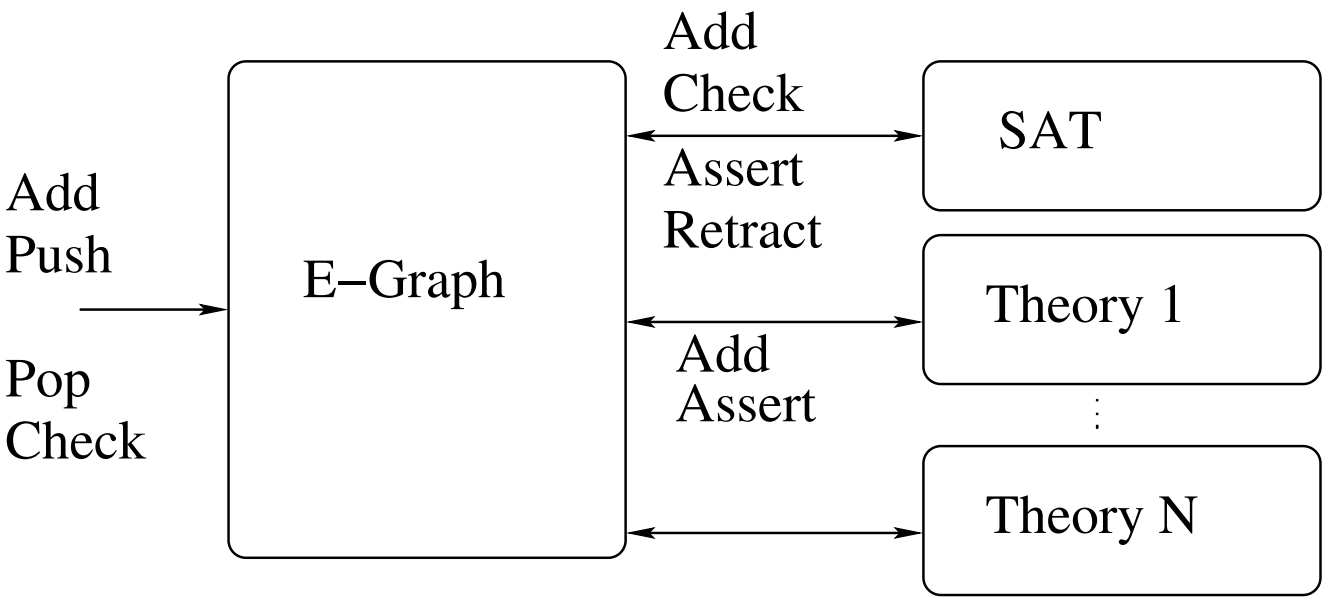
Critique of SMT Architectures

- **Mismatched interface:** The interface needs access to the information in the e-graph, not the SAT solver.
- **Redundant information:** Equality reasoning is done within both the e-graph and the theory solvers.
- **Possible race conditions:** Propagation of atoms from e-graph to theories and back is buffered, and information needed to simplify an atom might be in one of the queues.
- **Lack of contextual simplification:** No easy way to simplify the input relative to the known facts to reduce the size of the term universe.
- **Lack of multiple contexts:** Applications like PVS require multiple SMT contexts with cloning.

E(SAT) Architecture

- Atoms are treated as Boolean terms.
- All clauses and literals are asserted to the e-graph.
- The e-graph contains nodes for the entire term universe excluding clauses. This similar to Simplify.
- The SAT solver is a satellite theory.
- Clauses are processed by the SAT solver which also initiates splitting and backtracking.
- The e-graph maintains the global context with a single propagation queue.
- Bit-vectors are just tuples of Booleans, so a bit-vector solver is not needed.

E(SAT) Architecture



Advantages of E(SAT)

- **Contextual simplification:** Input clauses, literals, and terms can be simplified relative to the e-graph.
- **Absence of race conditions:** There is only one propagation queue.
- **E-graph propagation:** Equality information can be used to detect new equalities between terms.
- **Model building:** The e-graph has all the information needed to build and maintain a consistent model.
- **Proof:** All the explanation information needed to construct proofs can be kept in the e-graph.

E-Graph Propagation

- The e-graph can be augmented with propagation rules, e.g.,
  1. $(a_1, \ldots, a_n) \sim (b_1, \ldots, b_n) \implies a_i \sim b_i$
  2. $\ldots \implies \mathtt{IF}(a, b, c) \sim \ldots$
  3. $a \sim b \implies \mathrm{eq}(a, b) \sim \mathtt{TRUE}$
  4. $cons(a_1, a_2) \sim cons(b_1, b_2) \implies a_i \sim b_i$
  5. $(p_1 \vee \ldots \vee p_n) \sim \mathtt{FALSE} \implies p_i \sim \mathtt{FALSE}$
- The e-graph can also initiate non-SAT case-splitting (as in Yices 0).

Coarse-Grained Integration: The Evidential Tool Bus

- Fine-grained integration as in SMT requires high-performance interaction between components in a single program.
- Large inference tasks require the coarse-grained cooperation between multiple tools.
- The Evidential Tool Bus ontology consists of files and judgements.
- Judgements can be
  1. Syntactic: C is the concrete syntax for A according to tool τ, or
  2. Semantic: A is the abstraction of C with respect to predicates π according to tool τ
- We take a minimalist approach to language standardization, but advocate formally justified translations between different notations.

The Kernel of Truth

- Deduction can be carried out by rigorous formal rules of inference.
- With mechanization, we can, in principle, achieve nearly absolute certainty, but in practice, there are many gaps.
- *How can we combine a high degree of automation in verification tools while retaining trust?*
- *Check the verification, but verify the checker.*
- *The Kernel of Truth* contains verified checkers whose verifications have been checked.

Did I Ever Tell You How Lucky You are? [Dr. Seuss]

Oh, the jobs people work at!
Out west, near Hawtch-Hawtch,
there's a Hawtch-Hawtcher Bee-Watcher.
His job is to watch ...
is to keep both his eyes on the lazy town bee.
A bee that is watched will work harder, you see.
Well ... he watched and he watched.
But, in spite of his watch,
that bee didn't work any harder. Not mawtch.
So then somebody said,
"Our old bee-watching man
just isn't bee-watching as hard as he can.
He ought to be watched by another Hawtch-Hawtcher.
The thing that we need
is a Bee-Watcher-Watcher."
WELL ...

Did I Ever Tell You How Lucky You are? [Dr. Seuss]

. . .

The Bee-Watcher Watcher watched the Bee-Watcher.
He didn't watch well. So another Hawtch-Hawtcher
had to come in as a Watch-Watcher-Watcher.
And today all the Hawtchers who live in Hawtch-Hawtch
are watching on Watch-Watcher-Watchering-Watch,
Watch-Watching the Watcher who's watching that bee.
You're not a Hawtch-Hawtcher. You're lucky you see.

N. G. de Bruijn on Trust

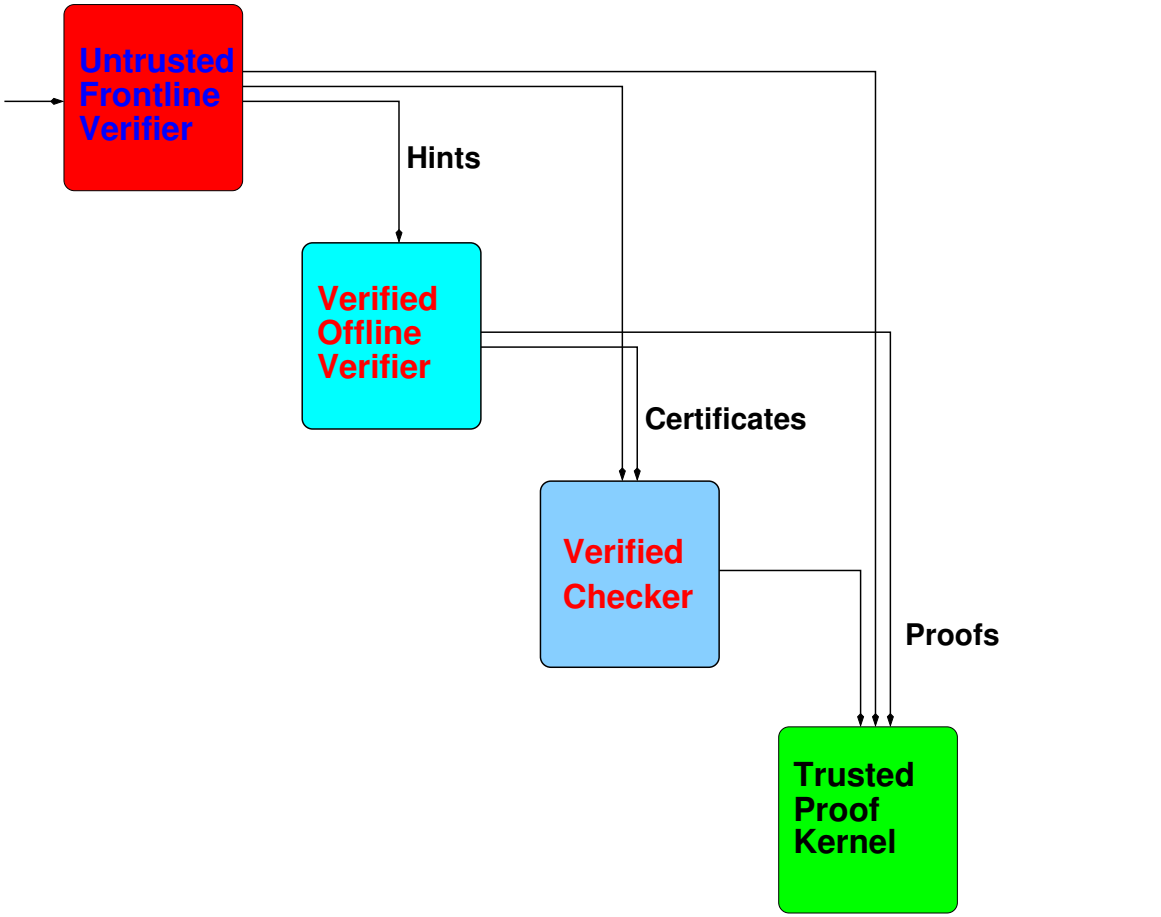
. . . we ask whether this guarantee would be weakened by leaving the mechanical verification to a machine. This is a very reasonable, relevant and important question. It is related to proving the correctness of fairly extensive computer programs, and checking the interpretation of the specifications of those programs. And there is more: the hardware, the operating system have to be inspected thoroughly, as well as the syntax, the semantics and the compiler of the programming language. And even if all this would be covered to satisfaction, there is the fear that a computer might make errors without indicating them by total breakdown.

I do not see how we ever can get to an absolute guarantee. But one has to admit that compared to human mechanical verification, computers are superior in every respect.

Proof Generation to Verified Inference Procedures

- There are a spectrum of options for achieving trust.
- At one extreme, we can *generate* formal proofs that are validated by a primitive proof checker.
- This kernel proof checker and its runtime environment will have to be trusted.
- *Proof generation imposes a serious time, space, effort overhead.*
- At the other extreme, we can *verify* the inference procedure by proving that every claim has a proof.
- We have to recursively/reflectively trust the inference procedures used in this verification.
- *Verifying cutting-edge inference tools is a fool's errand.*

Kernel of Truth



A Hierarchy of Checkers

- Many inference tools can have their claims certified relative to other inference tools.
- For example, the computations of a BDD package can be certified by a SAT solver.
- Similarly, a static analysis tool can be certified by an SMT solver.
- An SMT solver can itself be certified using a SAT solver and certificate checkers for the individual theories.
- A SAT solver can be certified by generating resolution proofs.
- But we can also have verified reference tools, like a verified SAT solver.
- Claims that are reducible to a common foundation can be shared across different systems.

Certifying PicoSAT

- PicoSAT generates resolution traces in a specific DIMACS-like format.
- With Andrei Dan and Antoine Toubhans, we have defined and verified a trace checker for these proofs in PVS.
- If the resolution proof derives the (possibly empty) clause κ from K, then $\vdash \overline{K}, \kappa$.

```
th_list:  THEOREM
     LET result: (tr_clause?) =
           resolution_list(lntcA) IN
      conclusion(proof_th_list(lntcA)) =
        append(not_or_map(lntcA),
               translate_clause(result))
      AND checkProof(empty_seq)(proof_th_list(lntcA))
```

- The trace checker is executable.

Conclusions

- Inference systems should be modular.
- And the software architecture should reflect this modularity.
- But this is easier said than done.
- We have outlined architectural designs for both fine-grained and coarse-grained composition of inference procedures.
- We have also presented an architecture for trust in inference.
- Many of these ideas and principles are applicable to software in general, but are critical for inference tools.

Verified Software: Theories, Tools, and Experiments (August 16–19, 2010)

- Please attend the VSTTE conference next month. For details, see

  `http://www.macs.hw.ac.uk/vstte10/`

- It runs alongside the incredible Edinburgh Festival.
- Students can essentially attend it for free.
- We will have an informal verification competition during the meeting.