

Decision Procedures for Automated Verification

Alessandro Armando

armando@dist.unige.it

Università di Genova



5 Settembre 2006

Scuola Estiva di Logica
Palazzo Feltrinelli, Gargnano

- 1 **Formal Verification**
- 2 Decision Procedures for Verification
 - Decision procedures for propositional logic
 - Deciding sets of equalities
 - Deciding sets of linear (arithmetic) constraints
- 3 Combining Satisfiability Procedures
- 4 Integrating Satisfiability Procedures
- 5 Encoding Techniques
 - C Bounded Model Checking

Verification is the act of proving or disproving whether a (hardware or software) system enjoys a certain property.

A verification technique: Testing

- **Testing** is the most widely used verification technique.
- It amounts to checking whether the system behaves correctly on a **finite** subset of all possible inputs;
- It is a well established and powerful verification technique

A verification technique: Testing

- **Testing** is the most widely used verification technique.
- It amounts to checking whether the system behaves correctly on a **finite** subset of all possible inputs;
- It is a well established and powerful verification technique

A verification technique: Testing

- **Testing** is the most widely used verification technique.
- It amounts to checking whether the system behaves correctly on a **finite** subset of all possible inputs;
- It is a well established and powerful verification technique

- **Testing** is the most widely used verification technique.
- It amounts to checking whether the system behaves correctly on a **finite** subset of all possible inputs;
- It is a well established and powerful verification technique,

BUT ...

Limitations of traditional Verification techniques: an example

October 30, 1994:

- A bug in the Intel Pentium:

$x := 4195835$

$y := 3145727$

$z := x - (x/y) * y$

- The chip gave as answer $z = 256$.
- Intel was forced to offer to replace all flawed Pentium processors.



Limitations of traditional Verification techniques: an example

October 30, 1994:

- A bug in the Intel Pentium:

$$x := 4195835$$

$$y := 3145727$$

$$z := x - (x/y) * y$$

- The chip gave as answer $z = 256$.
- Intel was forced to offer to replace all flawed Pentium processors.



Limitations of traditional Verification techniques: an example

October 30, 1994:

- A bug in the Intel Pentium:

$$x := 4195835$$

$$y := 3145727$$

$$z := x - (x/y) * y$$

- The chip gave as answer $z = 256$.
- Intel was forced to offer to replace all flawed Pentium processors.



Limitations of traditional Verification techniques: another example

June 4, 1996:

- The ESA rocket Ariane 5 exploded just 40 seconds after lift-off.
- The destroyed rocket and its cargo were evaluated \$500 million.
- **Cause:** a 64 bit floating point number relating to the horizontal velocity of the rocket w.r.t. platform was converted to a 16 bit signed integer.



Limitations of traditional Verification techniques: another example

June 4, 1996:

- The ESA rocket Ariane 5 exploded just 40 seconds after lift-off.
- The destroyed rocket and its cargo were evaluated \$500 million.
- **Cause:** a 64 bit floating point number relating to the horizontal velocity of the rocket w.r.t. platform was converted to a 16 bit signed integer.



Limitations of traditional Verification techniques: yet another example

From `www.risks.org`:

Russian ATM software error
<`morten.krog@no.ey.com`>

Mon, 28 Aug 2006 12:17:41 +0200

A few days ago in Ekaterinburg city, in the Ural region in Russia, a man deposited 2000 rubles (\$74 USD) in an ATM. Sounds ordinary so far, however the ATM credited his account with 2 billion rubles (yes, *billion*, with a B). [...] all the banks' ATMs are turned off. No word yet on when they will be back up.



As the complexity of hardware and software systems increases, the limitations of traditional verification techniques (e.g. testing) become evident.

This is an increasingly serious problem for hardware and software vendors.

***Formal Verification** is the act of proving or disproving whether a (hardware or software) system enjoys a certain property, using formal methods (e.g. logic, automata, etc.).*

If successful, formal verification ensures that the system complies with its specification on **all** possible inputs.

It consists of two activities:

- 1 **Formalisation**
- 2 **Proving**

- Both the system and the property must be formalised, i.e. specified in a formal language enjoying a mathematically precise semantics.
- The result of the formalisation activity yields a problem of the form:

$$\mathcal{T}, M \models \phi$$

where

- M is a formal specification of the system under consideration (a set of logical formulae, the formal representation of a state machine),
- ϕ is a formula encoding the expected property, and
- \mathcal{T} is a *background theory*

Formal Verification: Proving

- 1 Formally proving or disproving that the system enjoys the expected property.
- 2 Can be **reduced** to the problem of determining whether ϕ logically follows from M and \mathcal{T} , i.e.

$$\mathcal{T}, M \vdash \phi$$

where \vdash is the derivability relation in some given logical calculus.

- 3 If carried out manually it can be (and usually is) a daunting task for all verification problems of practical interest.



Automated Theorem Proving

- 1 Formally proving or disproving that the system enjoys the expected property.
- 2 Can be **reduced** to the problem of determining whether ϕ logically follows from M and \mathcal{T} , i.e.

$$\mathcal{T}, M \vdash \phi$$

where \vdash is the derivability relation in some given logical calculus.

- 3 If carried out manually it can be (and usually is) a daunting task for all verification problems of practical interest.



Automated Theorem Proving

Problem: How to automate (at least in part) the activity of proving that a system enjoys a given property?

- **Uniform Proof Procedures**, i.e. proof procedures for first-order logic, or even higher-order logics
 - :-) Generality
 - :-(Does not guarantee termination on fragments known to be decidable
 - :-(Brittle
- **Decision Procedures**, i.e. procedures able to solve a given (decidable) logical problem in a finite amount of time.
 - :-) Efficient and, obviously, terminating
 - :-(Limited scope of applicability

Dilemma: Decision procedures or Uniform proof procedures?

This dilemma dominates the scene of Automated Theorem Proving since its early days!

- Excellent (albeit biased) survey of the debate in the invited talk by N. Shankar (SRI) at 3rd Federated Logic Conference (FLoC 2002).
- Here we give some excerpts.

Little Engines of Proof^a

N. Shankar

shankar@csl.sri.com

URL: <http://www.csl.sri.com/~shankar/>

Computer Science Laboratory
SRI International
Menlo Park, CA

^aSupported by NSF Grant CCR-0082560, DARPA/AFRL Contract F33615-00-C-3043, NASA Contract NAS1-00079, and SRI International. Opinions expressed are those of the author. Companion papers (with citations) appear in RTA'02 and FME'02, and on the web page above.

Big Engines or Little Engines

The metric here is the **scale of ambition**, not size or complexity.

The *field* of automated deduction took a wrong turn in the 1960s through an overarching emphasis on *big iron*, i.e., uniform first-order proof engines such as **resolution**.

Most applications of deduction are better served by some combination of little, domain-specific decision procedures.

Little engines have been gathering steam lately.

FLoC'02 represents a turning point in the big engines/little engines debate.

This talk motivates the construction and use of little engines of proof as a **grand challenge** for the 21st century.

Early Echoes of the Debate

Early History of Automated Reasoning

1954: Martin Davis programs a Presburger Arithmetic decision procedure.

Its great triumph was to prove that the sum of two even numbers is even.

Martin Davis

1957: Newell, Shaw, and Simon's logic theorist (LT): Introduced subgoaling, substitution, replacement, and forward and backward chaining, with *human*-oriented heuristics. Applied to theorems from Russell & Whitehead's *Principia Mathematica*.

Many early papers are collected in *Automated Reasoning: Vols. 1 & 2*, edited by Siekmann and Wrightson.

The Handbook of Automated Reasoning, edited by Robinson and Voronkov, is a good modern summary.

Wang versus Newell–Shaw–Simon

1958-60: Hao Wang showed that many LT proofs (and others from Russell/Whitehead) were in *easily* decidable fragments: **propositional logic**, **Bernays–Schönfinkel**. Hundreds of these theorems could be proved in minutes.

The most interesting lesson from these results is perhaps that even in a fairly rich domain, the theorems actually proved are mostly ones which call on a very small portion of the available resources of the domain. —Hao Wang

Wang versus Newell–Shaw–Simon

The controversy referred to may be succinctly characterized as being between the two slogans: “Simulate people” and “Use mathematical logic”. . . . Thus as early as 1961 Minsky remarked

. . . it seems clear that a program to solve real mathematical problems will have to combine the mathematical sophistication of Wang with the heuristic sophistication of Newell, Shaw, and Simon.

—Martin Davis

Due to the prevailing fashions, Wang’s ideas were ignored, but . . .

Wang Was Right. And How!

Hao Wang's Programme: Inferential Analysis

In contrast with pure logic, the chief emphasis of inferential analysis is on the efficiency of algorithms, which is usually obtained by paying a great deal of attention to the detailed structure of problems and their solutions, to take advantage of possible systematic short cuts.

:

That proof procedures for elementary logic can be mechanized is familiar. In practice, however, were we slavishly to follow these procedures without further refinements, we should encounter a prohibitively expansive element. . . . In this way we are led to a closer study of reduction procedures and of decision procedures for special domains, as well as of proof procedures of more complex sorts.

—Hao Wang

The Big Engine Dogma

Naïve dogma: *First-order logic is a general language for expressing mathematics ergo uniform first-order proof search is the right mechanism for automating mathematics.*

Naïve dogma has few takers.

Sophisticated dogma: *Uniform first-order proof search is the right framework for building in domain-specific automation.*

No evidence for its validity either.

Resolution-based methods (e.g., Otter) have been successful in settling hundreds of open problems in diverse areas of mathematics.

But, it is more of a temperamental diva than a trusted lieutenant.

Macrologic versus Micrologic

- Macrologic addresses the automation of a logic as a whole by means of some uniform proof method.
- Resolution is an example of a macrological proof method.
- Micrologic examines the kinds of automation needed to attack individual classes of problems.
- Decision procedures, model checkers, specific sets of rewrite rules are examples of micrological methods, i.e., little engines.
- Macrological methods are interesting and occasionally useful, but micrologic gets the unglamorous work done.

- 1 Formal Verification
- 2 Decision Procedures for Verification
 - Decision procedures for propositional logic
 - Deciding sets of equalities
 - Deciding sets of linear (arithmetic) constraints
- 3 Combining Satisfiability Procedures
- 4 Integrating Satisfiability Procedures
- 5 Encoding Techniques
 - C Bounded Model Checking

Satisfiability Procedures vs Decision Procedures

- Let \mathcal{T} be a decidable theory.
- Let ϕ be an arbitrary formula in the same language as \mathcal{T} .
 - We say that ϕ is *\mathcal{T} -satisfiable* iff $\mathcal{T} \cup \{\phi\}$ is satisfiable.
 - The problem of deciding the \mathcal{T} -satisfiability of ϕ is the *decision problem for \mathcal{T}* .
 - Any algorithm capable to solve the decision problem for \mathcal{T} is a *decision procedure for \mathcal{T}* .
- Let S be a set of literals in the same language as \mathcal{T} .
 - We say that S is *\mathcal{T} -satisfiable* iff $\mathcal{T} \cup S$ is satisfiable.
 - The problem of deciding the \mathcal{T} -satisfiability of S is the *satisfiability problem for \mathcal{T}* .
 - Any algorithm capable to solve the decision satisfiability problem for \mathcal{T} is a *satisfiability procedure for \mathcal{T}* .

Syntax:

- 1 A (propositional) atom (A_1, A_2, \dots) is a formula;
- 2 if ϕ_1 and ϕ_2 are formulae, then also $\neg\phi_1$, $(\phi_1 \wedge \phi_2)$, $(\phi_1 \vee \phi_2)$, $(\phi_1 \supset \phi_2)$, $(\phi_1 \leftrightarrow \phi_2)$ are formulae.

Semantics

- A (*propositional*) *assignment for ϕ* is a function $\mu : \text{Atoms}(\phi) \rightarrow \{T, F\}$.
- Assignments are extended to formulae in the following way:

ϕ_1	ϕ_2	$\neg\phi_1$	$(\phi_1 \wedge \phi_2)$	$(\phi_1 \vee \phi_2)$	$(\phi_1 \supset \phi_2)$	$(\phi_1 \leftrightarrow \phi_2)$
F	F	T	F	F	T	T
F	T	T	F	T	T	F
T	F	F	F	T	F	F
T	T	F	T	T	T	T

- A formula ϕ *is satisfiable* iff there exists an assignment μ for ϕ such that $\mu(\phi) = T$.

- The problem of deciding the satisfiability of a propositional formula is NP-complete [Cook, 1971].
 - The most important logical problems (validity, entailment, equivalence, ...) can be easily reduced to satisfiability, and are thus (co)NP-complete.
- ⇒ No existing worst-case-polynomial algorithm.

Truth Tables Method

Semantic Tableaux

Davis-Putnam-Longeman-Loveland (DPLL) Procedure

- *Key Idea:*
 - enumerate all possible assignments for ϕ
 - if $\mu(\phi) = T$ for some assignment μ , then return μ as satisfying assignment
 - otherwise return *Unsatisfiable*
- $2^{|\text{Atoms}(\phi)|}$ assignments must be considered in the worst case.
- Requires polynomial space
- Inefficient \Rightarrow seldom/never used in practice

- Search for an assignment satisfying ϕ .
- Applies recursively *elimination rules* to the connectives.
- If a branch contains A_i and $\neg A_i$ for some i , the branch is *closed*, otherwise it is *open*.
- If no rule can be applied to an open branch μ , then return μ ;
- If all branches are closed, the formula is not satisfiable;

Tableau Rules

$\frac{\phi_1 \wedge \phi_2}{\phi_1}$	$\frac{\neg(\phi_1 \vee \phi_2)}{\neg\phi_1}$	$\frac{\neg(\phi_1 \supset \phi_2)}{\phi_1}$	\wedge-elimination
ϕ_2	$\neg\phi_2$	$\neg\phi_2$	

$\frac{\neg\neg\phi}{\phi}$	\neg-elimination
-----------------------------	--------------------------------------

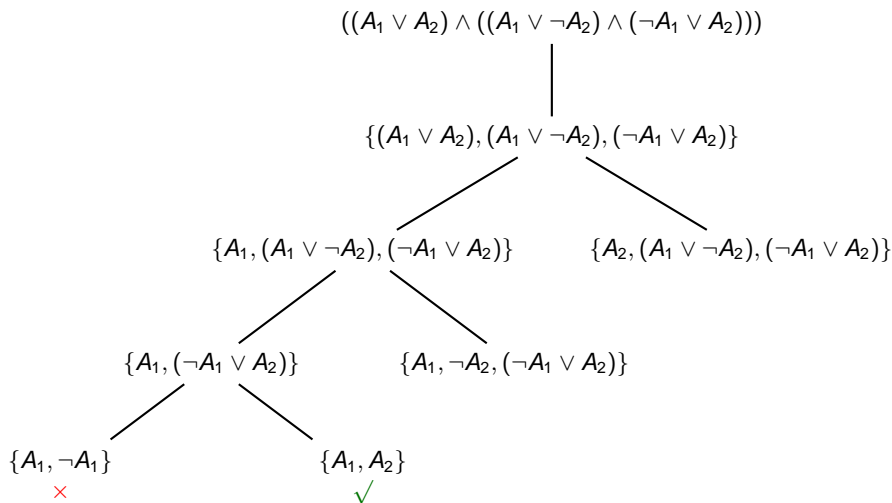
$\frac{\phi_1 \vee \phi_2}{\phi_1 \quad \phi_2}$	$\frac{\neg(\phi_1 \wedge \phi_2)}{\neg\phi_1 \quad \neg\phi_2}$	$\frac{\phi_1 \supset \phi_2}{\neg\phi_1 \quad \phi_2}$	\vee-elimination
--	--	---	--------------------------------------

$\frac{\phi_1 \leftrightarrow \phi_2}{\phi_1 \quad \neg\phi_1}$	$\frac{\neg(\phi_1 \leftrightarrow \phi_2)}{\phi_1 \quad \neg\phi_1}$	\leftrightarrow-elimination
$\phi_2 \quad \neg\phi_2$	$\neg\phi_2 \quad \phi_2$	

Tableau Algorithm

```
function Tableau( $\Gamma$ )
if  $A_i \in \Gamma$  and  $\neg A_i \in \Gamma$            /* branch closed */
  then return False;
if  $(\phi_1 \wedge \phi_2) \in \Gamma$              /*  $\wedge$ -elimination */
  then return Tableau( $\Gamma \cup \{\phi_1, \phi_2\} \setminus \{(\phi_1 \wedge \phi_2)\}$ );
if  $\neg\neg\phi \in \Gamma$                      /*  $\neg\neg$ -elimination */
  then return Tableau( $\Gamma \cup \{\phi\} \setminus \{\neg\neg\phi\}$ );
if  $(\phi_1 \vee \phi_2) \in \Gamma$            /*  $\vee$ -elimination */
  then return Tableau( $\Gamma \cup \{\phi_1\} \setminus \{(\phi_1 \vee \phi_2)\}$ ) or
  Tableau( $\Gamma \cup \{\phi_2\} \setminus \{(\phi_1 \vee \phi_2)\}$ );
:
return True;                               /* branch expanded */
```

Semantic Tableaux: An Example



Semantic Tableaux – Summary

- Handles all propositional formulas (CNF not required).
- Branches on disjunctions
- Intuitive, easy to extend
- Inefficient (compared to DPLL)
- Requires polynomial space

- ϕ must be in Conjunctive Normal Form (CNF), i.e. ϕ must be a conjunction of a disjunction of literals.
(A *literal* is an atom or a negated atom.)
- Given a formula ϕ , it is possible to build (in polynomial time) an equi-satisfiable formula ϕ' in CNF.
- Tries to build recursively an assignment μ satisfying ϕ .
- At each recursive step assigns a truth value to (all instances of) one atom.
- Performs deterministic choices first.

$$\frac{\phi \wedge (I)}{\phi[T/\ell]} \text{ (Unit)}$$

$$\frac{\phi}{\phi[T/\ell]} \text{ (Pure) if } \ell \text{ is a pure literal in } \phi$$

$$\frac{\phi}{\phi[T/\ell] \quad \phi[F/\ell]} \text{ (Split)}$$

- ℓ is a pure literal in ϕ iff ℓ occurs only positively in ϕ .
- *Split* is applied if and only if the other rules cannot be applied.

DPLL Algorithm

```
function DPLL( $\phi, \mu$ )
if  $\phi = T$                                 /* base */
    then return True;
if  $\phi = F$                                 /* backtrack */
    then return False;
if (a unit clause ( $\ell$ ) occurs in  $\phi$ )    /* Unit */
    then return DPLL( $\phi[T/\ell], \mu[T/\ell]$ );
if (a literal  $\ell$  occurs pure in  $\phi$ )    /* Pure */
    then return DPLL( $\phi[T/\ell], \mu[T/\ell]$ );
 $l :=$ choose-literal( $\phi$ );                    /* Split */
then return DPLL( $\phi[T/\ell], \mu[T/\ell]$ ) or
    DPLL( $\phi[F/\ell], \mu[F/\ell]$ );
```

DPLL Algorithm

```
function DPLL( $\phi, \mu$ )
if  $\phi = T$                                 /* base */
    then return True;
if  $\phi = F$                                 /* backtrack */
    then return False;
if (a unit clause ( $\ell$ ) occurs in  $\phi$ )    /* Unit */
    then return DPLL( $\phi[T/\ell], \mu[T/\ell]$ );
if (a literal  $\ell$  occurs pure in  $\phi$ )     /* Pure */
    then return DPLL( $\phi[T/\ell], \mu[T/\ell]$ );
 $l := \text{choose-literal}(\phi);$              /* Split */
then return DPLL( $\phi[T/\ell], \mu[T/\ell]$ ) or
    DPLL( $\phi[F/\ell], \mu[F/\ell]$ );
```

DPLL Algorithm

```
function DPLL( $\phi, \mu$ )
if  $\phi = T$                                 /* base */
    then return True;
if  $\phi = F$                                 /* backtrack */
    then return False;
if (a unit clause ( $\ell$ ) occurs in  $\phi$ )    /* Unit */
    then return DPLL( $\phi[T/\ell], \mu[T/\ell]$ );
if (a literal  $\ell$  occurs pure in  $\phi$ )    /* Pure */
    then return DPLL( $\phi[T/\ell], \mu[T/\ell]$ );
 $l := \text{choose-literal}(\phi);$               /* Split */
then return DPLL( $\phi[T/\ell], \mu[T/\ell]$ ) or
    DPLL( $\phi[F/\ell], \mu[F/\ell]$ );
```

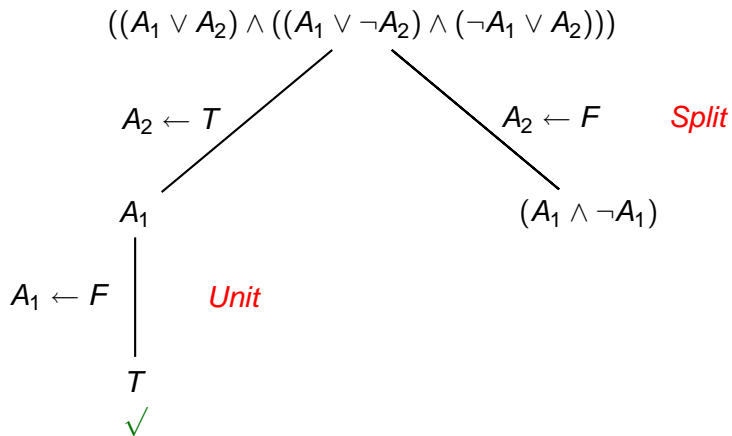
DPLL Algorithm

```
function DPLL( $\phi, \mu$ )
if  $\phi = T$                                 /* base */
    then return True;
if  $\phi = F$                                 /* backtrack */
    then return False;
if (a unit clause ( $\ell$ ) occurs in  $\phi$ )    /* Unit */
    then return DPLL( $\phi[T/\ell], \mu[T/\ell]$ );
if (a literal  $\ell$  occurs pure in  $\phi$ )    /* Pure */
    then return DPLL( $\phi[T/\ell], \mu[T/\ell]$ );
 $l :=$  choose-literal( $\phi$ );                 /* Split */
then return DPLL( $\phi[T/\ell], \mu[T/\ell]$ ) or
    DPLL( $\phi[F/\ell], \mu[F/\ell]$ );
```

DPLL Algorithm

```
function DPLL( $\phi, \mu$ )  
if  $\phi = T$  /* base */  
    then return True;  
if  $\phi = F$  /* backtrack */  
    then return False;  
if (a unit clause ( $\ell$ ) occurs in  $\phi$ ) /* Unit */  
    then return DPLL( $\phi[T/\ell], \mu[T/\ell]$ );  
if (a literal  $\ell$  occurs pure in  $\phi$ ) /* Pure */  
    then return DPLL( $\phi[T/\ell], \mu[T/\ell]$ );  
 $l :=$ choose-literal( $\phi$ ); /* Split */  
then return DPLL( $\phi[T/\ell], \mu[T/\ell]$ ) or  
            DPLL( $\phi[F/\ell], \mu[F/\ell]$ );
```

DPLL: An Example



Preprocessing: preprocess the input formula so that to make it easier to solve

Look-ahead: exploit information about the remaining search space

- unit propagation
- pure literal
- splitting heuristics
- forward checking

Look-back: exploit information about search which has already taken place

- Backjumping
- Learning

- Handles CNF formulas (non-CNF variant known).
- Branches on truth values
 - ⇒ all instances of an atom assigned simultaneously
- Postpones branching as much as possible.
- Currently the most efficient SAT algorithm
- Requires polynomial space
- `choose-literal()` critical for efficiency!
- Many very efficient implementations.

- **Variables:** x, y, z, \dots
- **Formulae:** *equalities* of the form $x = y$ where x and y are variables.
- **Decision Problem:** Let \mathcal{E} be the following set of formulae:
 - $\forall x. x = x$ (reflexivity)
 - $\forall x. \forall y. (x = y \supset y = x)$ (simmetry)
 - $\forall x. \forall y. \forall z. ((x = y \wedge y = z) \supset x = z)$ (transitivity)

Given a set S of equalities and negated equalities,

Is $S \cup \mathcal{E}$ satisfiable?

A satisfiability procedure for Equality

We consider sequents of the form $G; F; D$,
where G is a set of equalities and disequalities,
 $F : \text{Vars} \rightarrow \text{Vars}$,
 D is a set of disequalities

Inference Rules

Delete	$\frac{x = y, G; F; D}{G; F; D}$	if $F^*(x) = F^*(y)$
Merge	$\frac{x = y, G; F; D}{G; F'; D}$	if $F^*(x) \neq F^*(y)$ and $F' = \text{union}(F, x, y)$
Diseq	$\frac{x \neq y, G; F; D}{G; F; x \neq y, D}$	
Contrad	$\frac{G; F; x \neq y, D}{\perp}$	if $F^*(x) = F^*(y)$

where

$$F^*(x) = \begin{cases} x, & \text{if } F(x) = x \\ F^*(F(x)), & \text{otherwise} \end{cases}$$

$$\text{union}(F, x, y) = F[y'/x'],$$

with $x' = F^*(x)$ and
 $y' = F^*(y)$.

Theorem

A finite set S of equalities and negated equalities is \mathcal{E} -unsatisfiable iff \perp is derivable from $S; Id; \emptyset$.

Quantifier-free Linear Arithmetics

- **Variables:** x, x_1, x_2, \dots range over \mathbb{Q} or \mathbb{R} (but can be extended to work over \mathbb{Z}).
- **Function symbols:** “+” for addition, unary “-”
Multiplication by a numeric constant n , i.e. nx , is allowed but it is a shorthand for $x + \dots + x$ with n occurrences of x .
- **Relational symbol:** “ \leq ” and “ $=$ ” (but $<$, $>$, \geq can be added)
- **Formulae (linear constraints):** $a_1x_1 + \dots + a_nx_n + c \leq 0$, where a_1, \dots, a_n are numeric constants.
- **Notation:** In place of $a_1x_1 + \dots + a_nx_n + c \leq 0$ we write $\mathbf{ax} + c \leq 0$, where $\mathbf{a} = [a_1, \dots, a_n]$ and $\mathbf{x} = [x_1, \dots, x_n]$.
Also, if $\mathbf{a} = [a_1, \dots, a_n]$, then \mathbf{a}_k denotes a_k for $k = 1, \dots, n$.

Key step:

$$\frac{\mathbf{ax} + \mathbf{c} \leq 0 \quad \mathbf{bx} + \mathbf{d} \leq 0}{\mathbf{b}_k(\mathbf{ax} + \mathbf{c}) + \mathbf{a}_k(\mathbf{bx} + \mathbf{d}) \leq 0} \text{ FM}(x_k) \quad \text{if } \mathbf{a}_k > 0 \text{ and } \mathbf{b}_k < 0$$

Facts:

- x_k does not occur in the conclusion of the rule
- The conclusion of the rule is logically equivalent (in the background theory) to $\exists x_k. (\mathbf{ax} + \mathbf{c} \leq 0 \wedge \mathbf{bx} + \mathbf{d} \leq 0)$.

Example:

$$\frac{2x_1 + x_2 + 1 \leq 0 \quad x_1 - 2x_2 - 1 \leq 0}{2(2x_1 + x_2 + 1) + 1(x_1 - 2x_2 - 1) \leq 0} \text{ FM}(x_2)$$

Key step:

$$\frac{\mathbf{ax} + \mathbf{c} \leq 0 \quad \mathbf{bx} + \mathbf{d} \leq 0}{\mathbf{b}_k(\mathbf{ax} + \mathbf{c}) + \mathbf{a}_k(\mathbf{bx} + \mathbf{d}) \leq 0} \text{ FM}(x_k) \quad \text{if } \mathbf{a}_k > 0 \text{ and } \mathbf{b}_k < 0$$

Facts:

- x_k does not occur in the conclusion of the rule
- The conclusion of the rule is logically equivalent (in the background theory) to $\exists x_k. (\mathbf{ax} + \mathbf{c} \leq 0 \wedge \mathbf{bx} + \mathbf{d} \leq 0)$.

Example:

$$\frac{2x_1 + x_2 + 1 \leq 0 \quad x_1 - 2x_2 - 1 \leq 0}{5x_1 + 1 \leq 0} \text{ FM}(x_2)$$

Fourier-Motzkin Elimination

```
function FM( $S$ )  
let  $S'$  be the result of replacing from  $S$  all formulae  
  of the form  $\mathbf{ax} + \mathbf{c} = 0$  with  $\mathbf{ax} + \mathbf{c} \leq 0 \wedge -\mathbf{ax} - \mathbf{c} \leq 0$ .  
while  $\text{Vars}(S') \neq \emptyset$  do  
  choose  $x_k \in \text{Vars}(S')$  do  
    eliminate from  $S'$  all the linear constraints  
      in which  $x_k$  occurs  
    and  
      replace them with all linear constraints  
        obtained by applying  $FM(x_k)$   
        to any pair of removed equations.  
  /* Now all constraints in  $S'$  are of the form  $c \leq 0$  */  
  if there exists  $c \leq 0 \in S'$  with  $c > 0$   
    then return Unsatisfiable;  
    else return Satisfiable;
```

- Used to determine the satisfiability of finite sets of linear constraints over \mathbb{Q} or \mathbb{R} .
- Conceptually simple and elegant.
- Many redundant intermediate linear constraints are generated.
- Phase I of the Simplex can be used for the same purpose and is far more efficient.

- **Theory of lists:**

$$\forall x, y. \text{car}(\text{cons}(x, y)) = x$$

$$\forall x, y. \text{cdr}(\text{cons}(x, y)) = y$$

$$\forall y. \text{cons}(\text{car}(y), \text{cdr}(y)) = y$$

- **Theory of arrays:** Let a be a variable of sort ARRAY, i and j of sort INDEX, and e of sort ELEM:

$$\forall a, i, e. \quad \text{select}(\text{store}(a, i, e), i) = e$$

$$\forall a, i, j, e. \quad (i \neq j \supset \text{select}(\text{store}(a, i, e), j) = \text{select}(a, j))$$

- **Theory of records: ...**

- **Theory of bit-vectors: ...**

⋮

- 1 Formal Verification
- 2 Decision Procedures for Verification
 - Decision procedures for propositional logic
 - Deciding sets of equalities
 - Deciding sets of linear (arithmetic) constraints
- 3 Combining Satisfiability Procedures**
- 4 Integrating Satisfiability Procedures
- 5 Encoding Techniques
 - C Bounded Model Checking

Combining Satisfiability Procedures

Problem: Given satisfiability procedures for two theories \mathcal{T}_1 and \mathcal{T}_2 how can we build a satisfiability procedure for $(\mathcal{T}_1 \cup \mathcal{T}_2)$?

Example: Let

- $P_{\mathcal{A}}$ be a satisfiability procedure for the theory of arrays (\mathcal{A}) and
- $P_{\mathcal{R}}$ be a satisfiability procedure for (rational) linear arithmetics (\mathcal{R}),

how can we determine the $(\mathcal{A} \cup \mathcal{R})$ -satisfiability of the following formula?

$$\text{select}(\text{store}(v, i, \text{select}(v, j)), i) \neq \text{select}(v, i) \wedge i+j \leq 2j \wedge j+4i \leq 5i$$

Combining Satisfiability Procedures

Problem: Given satisfiability procedures for two theories \mathcal{T}_1 and \mathcal{T}_2 how can we build a satisfiability procedure for $(\mathcal{T}_1 \cup \mathcal{T}_2)$?

Example: Let

- $P_{\mathcal{A}}$ be a satisfiability procedure for the theory of arrays (\mathcal{A}) and
- $P_{\mathcal{R}}$ be a satisfiability procedure for (rational) linear arithmetics (\mathcal{R}),

how can we determine the $(\mathcal{A} \cup \mathcal{R})$ -satisfiability of the following formula?

$$\text{select}(\text{store}(v, i, \text{select}(v, j)), i) \neq \text{select}(v, i) \wedge i+j \leq 2j \wedge j+4i \leq 5i$$

Decomposition: Example

Naive approach :

- Decompose ϕ into $\phi_{\mathcal{R}} \wedge \phi_{\mathcal{A}}$:

$$\phi_{\mathcal{R}} = i+j \leq 2j \wedge j+4i \leq 5i$$

$$\phi_{\mathcal{A}} = \text{select}(\text{store}(v, i, \text{select}(v, j)), i) \neq \text{select}(v, i)$$

- Apply $P_{\mathcal{R}}$ to $\phi_{\mathcal{R}}$:

\Rightarrow *satisfiable*

- Apply $P_{\mathcal{A}}$ to $\phi_{\mathcal{A}}$:

\Rightarrow *satisfiable*

- Return *satisfiable* ???

- In fact: *unsatisfiable*.

$$i+j \leq 2j \wedge j+4i \leq 5i \Rightarrow i = j$$

$$\text{select}(\text{store}(v, i, \text{select}(v, j)), i) \neq \text{select}(v, i) \wedge i = j \Rightarrow \\ \text{select}(v, i) \neq \text{select}(v, i)$$

- Problems :
 - Shared variables
 - Shared equality predicate
- Possible solution: *propagation of equalities between variables*

What if a formula contains symbols from both theories?

For example,

$\text{select}(\text{store}(v, i, \text{select}(v, j)), i) < \text{select}(v, i) \wedge i + j \leq 2j \wedge j + 4i \leq 5i$

Introduce new variables and decompose the formula as follows:

$x = \text{select}(\text{store}(v, i, \text{select}(v, j)), i) \wedge$

$y = \text{select}(v, i) \wedge$

$x < y \wedge$

$i + j \leq 2j \wedge$

$j + 4i \leq 5i$

What if a formula contains symbols from both theories?

For example,

$$\text{select}(\text{store}(v, i, \text{select}(v, j)), i) < \text{select}(v, i) \wedge i + j \leq 2j \wedge j + 4i \leq 5i$$

Introduce new variables and decompose the formula as follows:

$$x = \text{select}(\text{store}(v, i, \text{select}(v, j)), i) \wedge$$

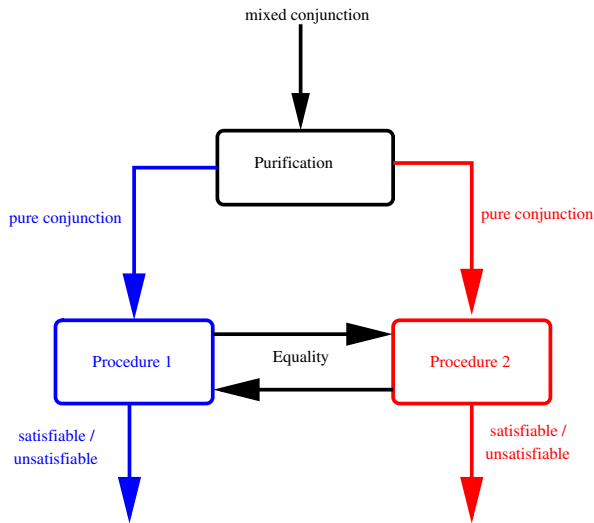
$$y = \text{select}(v, i) \wedge$$

$$x < y \wedge$$

$$i + j \leq 2j \wedge$$

$$j + 4i \leq 5i$$

Nelson-Oppen Approach: Picture



● **Nelson-Oppen Approach**

- Union of disjoint theories.
- Combination of satisfiability procedures.
- General brute-force method, easy to understand, but no much interest in practice.

● **Shostak Approach**

- Union of disjoint theories + some additional requirements.
- Combination of algorithm for congruence closure (for the theory of equality) and specific procedures for other theories.
- Efficient method, difficult to get right and prove correct, but implemented in many systems.

- 1 Formal Verification
- 2 Decision Procedures for Verification
 - Decision procedures for propositional logic
 - Deciding sets of equalities
 - Deciding sets of linear (arithmetic) constraints
- 3 Combining Satisfiability Procedures
- 4 Integrating Satisfiability Procedures
- 5 Encoding Techniques
 - C Bounded Model Checking

It is often useful/necessary to integrate satisfiability procedures with other reasoning components:

- with rewrite engines for simplifying expressions.
 - The integration of satisfiability procedures within the Boyer&Moore theorem prover is described in [Boyer and Moore, 1988].
 - The integration of satisfiability procedures within the Maple computer algebra system is described in [Weibel and Gonnet].
 - A general integration schema for satisfiability procedures with rewriting is described in [Armando and Ranise, 2003].
- with decision procedures for propositional logic to get decision procedures for the quantifier-fragments of the theory.

Integrating satisfiability procedures with decision procedures for propositional logic

We are interested in deciding the satisfiability/validity of formulae of the form:

- 1 An atomic formula of \mathcal{T} is a formula;
- 2 if ϕ_1 and ϕ_2 are formulae, then also $\neg\phi_1$, $(\phi_1 \wedge \phi_2)$, $(\phi_1 \vee \phi_2)$, $(\phi_1 \supset \phi_2)$, $(\phi_1 \leftrightarrow \phi_2)$ are formulae.

where \mathcal{T} is a decidable theory for which a satisfiability procedure is available.

Integrating satisfiability procedures with the Semantic Tableaux

```
function  $\mathcal{T}$ -Tableau( $\Gamma$ )  
if  $A_i \in \Gamma$  and  $\neg A_i \in \Gamma$                                 /* branch closed */  
  then return False;  
if  $(\phi_1 \wedge \phi_2) \in \Gamma$                                     /*  $\wedge$ -elimination */  
  then return  $\mathcal{T}$ -Tableau( $\Gamma \cup \{\phi_1, \phi_2\} \setminus \{(\phi_1 \wedge \phi_2)\}$ );  
if  $\neg\neg\phi \in \Gamma$                                            /*  $\neg\neg$ -elimination */  
  then return  $\mathcal{T}$ -Tableau( $\Gamma \cup \{\phi\} \setminus \{\neg\neg\phi\}$ );  
if  $(\phi_1 \vee \phi_2) \in \Gamma$                                     /*  $\vee$ -elimination */  
  then return  $\mathcal{T}$ -Tableau( $\Gamma \cup \{\phi_1\} \setminus \{(\phi_1 \vee \phi_2)\}$ ) or  
     $\mathcal{T}$ -Tableau( $\Gamma \cup \{\phi_2\} \setminus \{(\phi_1 \vee \phi_2)\}$ );  
:  
return  $\mathcal{T}$ -satisfiable( $\Gamma$ );                                /* branch expanded */
```

Integrating satisfiability procedures with DPLL

```
function  $\mathcal{T}$ -DPLL( $\phi, \mu$ )
if  $\phi = T$                                 /* base */
  then return  $\mathcal{T}$ -satisfiable( $\mu$ );
if  $\phi = F$                                 /* backtrack */
  then return False;
if (a unit clause ( $\ell$ ) occurs in  $\phi$ )    /* Unit */
  then return  $\mathcal{T}$ -DPLL( $\phi[T/\ell], \mu[T/\ell]$ );
 $l := \text{choose-literal}(\phi)$ ;            /* Split */
then return  $\mathcal{T}$ -DPLL( $\phi[T/\ell], \mu[T/\ell]$ ) or
          $\mathcal{T}$ -DPLL( $\phi[F/\ell], \mu[F/\ell]$ );
```

where $\mu^* = \{\alpha : \alpha \in \text{Atoms}(\phi) \text{ and } \mu(\alpha) = T\} \cup \{\neg\alpha : \alpha \in \text{Atoms}(\phi) \text{ and } \mu(\alpha) = F\}$.

Note: Pure Literal optimisation unsound here and therefore dropped.

- 1 Formal Verification
- 2 Decision Procedures for Verification
 - Decision procedures for propositional logic
 - Deciding sets of equalities
 - Deciding sets of linear (arithmetic) constraints
- 3 Combining Satisfiability Procedures
- 4 Integrating Satisfiability Procedures
- 5 **Encoding Techniques**
 - **C Bounded Model Checking**

C Bounded Model Checking

- Bounded Model Checking (basic idea):
 - 1 reduce a bounded model checking problem to a satisfiability problem in propositional logic (SAT), then
 - 2 use a state-of-the-art SAT solver to solve the problem.
- Initially applied successfully to analyze HW circuits
- Recently applied to find bugs in sequential programs (CBMC)
- However model checking of SW poses new challenges as programs often deal with large or potentially unbounded data.
- The CBCM approach to SW Model Checking can be adapted to use a SMT solver instead of a SAT solver.
- The usage of SMT solvers instead of SAT solvers improves performance considerably on many problems of practical interest.

C Bounded Model Checking

- Bounded Model Checking (basic idea):
 - 1 reduce a bounded model checking problem to a satisfiability problem in propositional logic (SAT), then
 - 2 use a state-of-the-art SAT solver to solve the problem.
- Initially applied successfully to analyze HW circuits
- Recently applied to find bugs in sequential programs (CBMC)
- However model checking of SW poses new challenges as programs often deal with large or potentially unbounded data.
- The CBCM approach to SW Model Checking can be adapted to use a SMT solver instead of a SAT solver.
- The usage of SMT solvers instead of SAT solvers improves performance considerably on many problems of practical interest.

- 1 Preprocessing
 - 1 Unwinding loops
 - 2 Turning the program in Single Assignment Form
 - 3 Turning the program in Conditional Normal Form
- 2 Encoding
- 3 Solving

Preprocessing: Unwinding Loops

Every loop is reduced to a sequence of k nested `if` statements + an additional *unwinding assertion*.

Example: If $k = 2$, then

```

:
i=0;
while(i<5) {
    a[i]=i;
    i++;
}
:
:
if(i<5) {
    a[i]=i;
    i++;
    if(i<5) {
        a[i]=i;
        i++;
        assert(!i<5);
    }
}
:
```



Preprocessing: Single Assignment Form

Rename program variables in such a way that each variable is assigned exactly once.

Example:

```
i = a[0];  
if(x>0) {  
    if(x<10)  
        x=x+1;  
    else  
        x=x-1;  
}  
assert(y>0 && y<5);  
a[y]=i;
```

⇒

```
i1 = a0[0];  
if(x0>0) {  
    if(x0<10)  
        x1=x0+1;  
    else  
        x2=x1-1;  
}  
assert(y0>0 && y0<5);  
a1[y0]=i1;
```

Preprocessing: Conditional Normal Form

Remove **else** constructs and push **if** statements downwards in the abstract syntax tree of the program until they are applied to atomic statements only.

Example:

```
i1 = a0[0];
if(x0>0) {
    if(x0<10)
        x1=x0+1;
    else
        x2=x1-1;
}
assert(y0>0 && y0<5);
a1[y0]=i1;
```

⇒

```
if(true) i1 = a0[0];
if(x0>0 && x0<10) x1=x0+1;
if(x0>0 && !(x0<10)) x2=x1-1;
if(true) assert(y0>0 && y0<5);
if(true) a1[y0]=i1;
```

- Let P be the program in conditional normal form resulting from the application of the previous transformations to the input program.
- Let \mathcal{T} the union of the theory of arrays and the theory of bit-vectors.
- We build two sets of quantifier-free formulae \mathcal{C} and \mathcal{P} such that $\mathcal{T}, \mathcal{C} \models \bigwedge \mathcal{P}$ if and only if no computation path of P violates any **assert** statement in P .

Example:

```
if(true) i1 = a0[0];  
if(x0>0 && x0<10) x1=x0+1;  
if(x0>0 && !(x0<10)) x2=x1-1;   
if(true) assert(y0>0 && y0<5);  
if(true) a1[y0]=i1;
```

$$\begin{aligned} \mathcal{C} &= \{ i_1 = (T ? \text{select}(a_0, 0) : i_0), \\ & \quad x_1 = ((x_0 > 0 \wedge x_0 < 10) ? x_0 + 1 : x_0), \\ & \quad x_2 = ((x_0 > 0 \wedge \neg(x_0 < 10)) ? x_1 - 1 : x_1), \\ & \quad a_1 = (T ? \text{store}(a_0, y_0, i_1) : a_0) \} \\ \mathcal{P} &= \{ T \supset (y_0 > 0 \wedge y_0 < 5) \} \end{aligned}$$

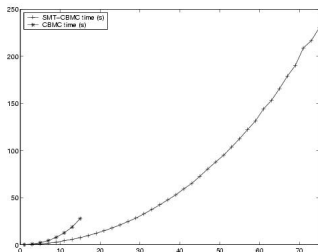
$v = (c ? e_1 : e_2)$ abbreviates the formula $(c \supset v = e_1) \wedge (\neg c \supset v = e_2)$.

- In order to check whether $\mathcal{C} \models_{\mathcal{T}} \bigwedge \mathcal{P}$, we can use any SMT solver capable to decide the union of the theory of arrays and the theory of bit-vectors.
 - \implies SMT-CBMC [Armando et al., 2006] uses CVC Lite as SMT-solver.
- If we regard arrays as a finite collection of distinct memory cells, $\mathcal{C} \models_{\mathcal{T}} \bigwedge \mathcal{P}$ can be reduced to a (usually very big) propositional formula that can be solved by a SAT solver.
 - \implies CBMC [Clarke et al., 2004] uses Chaff as SAT-solver.

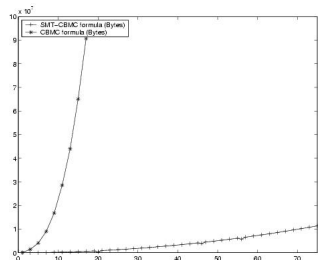
Experiments

```
int a[N]={N-1,...,0};
void main() {
  int i;
  SelectSort();
  for(i=0;i<N-1;i++){
    assert(a[i]<=a[i+1]);
  }
}
```

```
void SelectSort() {
  int i,j,t,min;
  for(j=0;j<N-1;j++) {
    min=j;
    for(i=j+1; i<N; i++)
      if(a[i]>a[min])
        min=i;
    t=a[j];
    a[j]=a[min];
    a[min]=t;
  }
}
```



(a) Time (seconds) spent by the tools



(b) Size of the formulae

- **Combination Methods in Automated Reasoning**
<http://combination.cs.uiowa.edu/>
- **SMT-LIB - The Satisfiability Modulo Theories Library**
<http://combination.cs.uiowa.edu/>
- **SMT-COMP - The Satisfiability Modulo Theories Library**
<http://www.csl.sri.com/users/demoura/smt-comp/index.shtml>
- **SATLive! - Up-to-date links for SAT**
<http://goedel.cs.uiowa.edu/smtlib/>
- **SATLIB - The Satisfiability Library**
<http://www.satlive.org/index.jsp>
- **Web Forum on Automated Reasoning**
<http://www-users.cs.york.ac.uk/~frisch/AR-Forum/>

- The slides by Silvio Ranise (available at the <http://www.loria.fr/~ranise/>) are a good source of information about combination methods for satisfiability procedures.
- The slides by Roberto Sebastiani (available at the <http://dit.unitn.it/~rseba/>) are a good source of information about SAT-solvers in general and about the integration of satisfiability procedures with SAT-solvers.

References

- Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In Antti Valmari, editor, *SPIN*, volume 3925 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2006. ISBN 3-540-33102-6. URL http://dx.doi.org/10.1007/11691617_9.
- Alessandro Armando and Silvio Ranise. Constraint contextual rewriting. *Journal of Symbolic Computation*, 36(1–2):193–216, July/August 2003. ISSN 0747-7171.
- R.S. Boyer and J.S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. *Machine Intelligence*, 11:83–124, 1988.
- Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of TACAS04*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004. ISBN 3-540-21299-X.
- S. A. Cook. The complexity of theorem proving procedures. In *3rd Annual ACM Symposium on the Theory of Computation*, pages 151–158, 1971.
- Trudy Weibel and Gaston H. Gonnet. An assume facility for CAS, with a sample implementation for Maple. In *Proceedings of DISCO '92*, pages 95–103.