Formalization of SAT Solvers

Filip Marić*

*Faculty of Mathematics
University of Belgrade

Second Workshop on Formal and Automated Theorem Proving and Applications

Overview

SAT problem

Definition (SAT problem)

Propositional satisfiability (SAT) problem is the problem of deciding if there is a truth assignment under which a given propositional formula (in CNF) evaluates to true. Satisfying truth assignment is a model of the formula.

Example

The formula

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3)$$

is true in the model $\{x_1, \neg x_2, \neg x_3\}$.

Example

The formula

$$(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_1) \wedge x_1$$

is not satisfiable.

Applications of SAT solving

Many practical problems can be encoded in SAT.

- Electronic Design Automation
- Software and Hardware Verification
- Artificial Intelligence
- Planing and Scheduling
- Operations Research

SAT Solving Algorithms

Complete algorithms - for every SAT instance can either find its model or show that no model exists.

Stohastic algorithms - cannot show that no model exists, but can find a model of some large SAT instances very quickly.

We are only interested in complete algorithms.

SAT Solving Algorithms

Complete algorithms - for every SAT instance can either find its model or show that no model exists.

Stohastic algorithms - cannot show that no model exists, but can find a model of some large SAT instances very quickly.

We are only interested in complete algorithms.

SAT Solving Algorithms

Complete algorithms - for every SAT instance can either find its model or show that no model exists.

Stohastic algorithms - cannot show that no model exists, but can find a model of some large SAT instances very quickly.

We are only interested in complete algorithms.

Classic DPLL - recursive formulation

```
function dpll (F : Formula) : (SAT, UNSAT)
begin
  if F is empty then                                        BASE
    return SAT
  else if there is an empty clause in F then
    return UNSAT
  else there is a unit clause [l] in F then            INFERENCE
    return dpll(F[l → ⊤])
  else if there is a pure literal l in F then
    return dpll(F[l → ⊤])
  else begin                                              SEARCH
    select a literal l occurring in F
    if dpll(F[l → ⊤]) = SAT then
      return SAT
    else
      return dpll(F[l → ⊥])
  end
end
```

Classic DPLL - recursive formulation

```
function dpll (F : Formula) : (SAT, UNSAT)
begin
  if F is empty then                                   BASE
    return SAT
  else if there is an empty clause in F then
    return UNSAT
  else there is a unit clause [l] in F then       INFERENCE
    return dpll(F[l → ⊤])
  else if there is a pure literal l in F then
    return dpll(F[l → ⊤])
  else begin                                         SEARCH
    select a literal l occurring in F
    if dpll(F[l → ⊤]) = SAT then
      return SAT
    else
      return dpll(F[l → ⊥])
  end
end
```

Classic DPLL - recursive formulation

```
function dpll (F : Formula) : (SAT, UNSAT)
begin
  if F is empty then                                          BASE
    return SAT
  else if there is an empty clause in F then
    return UNSAT
  else there is a unit clause [l] in F then              INFERENCE
    return dpll(F[l → ⊤])
  else if there is a pure literal l in F then
    return dpll(F[l → ⊤])
  else begin                                                 SEARCH
    select a literal l occurring in F
    if dpll(F[l → ⊤]) = SAT then
      return SAT
    else
      return dpll(F[l → ⊥])
  end
end
```

Classic DPLL - recursive formulation

```
function dpll (F : Formula) : (SAT, UNSAT)
begin
  if F is empty then                                          BASE
    return SAT
  else if there is an empty clause in F then
    return UNSAT
  else there is a unit clause [l] in F then              INFERENCE
    return dpll(F[l → ⊤])
  else if there is a pure literal l in F then
    return dpll(F[l → ⊤])
  else begin                                                 SEARCH
    select a literal l occurring in F
    if dpll(F[l → ⊤]) = SAT then
      return SAT
    else
      return dpll(F[l → ⊥])
  end
end
```

Progress in SAT Solving

- Spectacular improvements in the last decade.
- Possible to solve formulae with $\approx$ 10 000 variables and $\approx$ 1 000 000 clauses

Reasons for this success

Conceptual enhancements of the DPLL procedure

- *backjumping*
- *conflict-driven lemma learning*
- *restarts*

Better implementation

- non-recursive implementation
- smart data-structures
- *two-watched literals scheme* for unit propagation,

Heuristic components

- *literal selection strategies*

Motivation

Goal

Have trusted SAT solvers.

Approaches

1. Make SAT solvers produce proofs of their claims and verify those proofs by independent trusted checkers.
2. Apply formal methods and verify SAT solvers themselves.

Motivation

Goal

Have trusted SAT solvers.

Approaches

1. Make SAT solvers produce proofs of their claims and verify those proofs by independent trusted checkers.
2. Apply formal methods and verify SAT solvers themselves.

Motivation

Goal

Have trusted SAT solvers.

Approaches

1. Make SAT solvers produce proofs of their claims and verify those proofs by independent trusted checkers.
2. Apply formal methods and verify SAT solvers themselves.

Descriptions of modern SAT solvers

Concrete descriptions - Usually given in a form of programming language (pseudo)code. Close to real implementations, but hard to understand and reason about.

Abstract descriptions - Usually given as state transition systems. Easy to understand, formalize and reason about, but hide many important implementation details.

Approaches for verification

1. Verify only abstract descriptions.
2. Use Hoare-logic style verification for imperative code.
3. Formalize and verify SAT solvers by shallow embedding into HOL and automatically extract executable code.

Overview

Syntax

Example

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3)$$

Model: $\{x_1, \neg x_2, \neg x_3\}$

Isabelle types

```
types     Variable  = nat
datatype Literal   = Pos Variable
                    | Neg Variable
types     Clause    = "Literal list"
types     Formula   = "Clause list"

types     Valuation = "Literal list"
```

Semantics

Definition

$v \vDash l$ if and only if $l \in v$

```
literalTrue :: "Literal => Valuation => bool"
```

$v \vDash\neg\, l$ if and only if $\bar{l} \in v$

```
literalFalse :: "Literal => Valuation => bool"
```

$v \vDash c$ if and only if $\exists l.\ l \in c \wedge v \vDash l$

```
clauseTrue :: "Clause => Valuation => bool"
```

$v \vDash\neg\, c$ if and only if $\forall l.\ l \in c \rightarrow v \vDash\neg\, l$

```
clauseFalse :: "Clause => Valuation => bool"
```

$v \vDash F$ if and only if $\forall c.\ c \in F \rightarrow v \vDash c$

```
formulaTrue :: "Formula => Valuation => bool"
```

$v \vDash\neg\, F$ if and only if $\exists c.\ c \in F \wedge v \vDash\neg\, c$

```
formulaFalse :: "Formula => Valuation => bool"
```

Semantics (cont.)

Definition

(consistent v) if and only if $(\neg\exists l.\ v \models l \wedge v \models \bar{l})$

```
consistent ::  "Valuation => bool"
```

(model v F) if and only if (consistent $v \wedge v \models F$)

```
model ::  "Valuation => Formula => bool"
```

(sat F) if and only if ($\exists v.$ model v F)

```
satisfiable ::  "Formula => bool"
```

Assertion trails

When building a non-recursive implementation the notion of valuation is extended.

Definition (Assertion trail)

Assertion trail is a list of literals, some of which are marked as *decision literals*. Decision literals split the trail into *levels*.

Example

A trail M could be $[+1, |-2, +6, |+5, -3, +4, |-7]$.

Assertion trails

When building a non-recursive implementation the notion of valuation is extended.

Definition (Assertion trail)

Assertion trail is a list of literals, some of which are marked as *decision literals*. Decision literals split the trail into *levels*.

Example

A trail M could be $[+1, |-2, +6, |+5, -3, +4, |-7]$.

Assertion trails

When building a non-recursive implementation the notion of valuation is extended.

Definition (Assertion trail)

Assertion trail is a list of literals, some of which are marked as *decision literals*. Decision literals split the trail into *levels*.

Example

A trail M could be $[+1, |-2, +6, |+5, -3, +4, |-7]$.

Assertion trails

When building a non-recursive implementation the notion of valuation is extended.

Definition (Assertion trail)

Assertion trail is a list of literals, some of which are marked as *decision literals*. Decision literals split the trail into *levels*.

Example

A trail M could be $[+1, |-2, +6, |+5, -3, +4, |-7]$.

Assertion trails

When building a non-recursive implementation the notion of valuation is extended.

Definition (Assertion trail)

Assertion trail is a list of literals, some of which are marked as *decision literals*. Decision literals split the trail into *levels*.

Example

A trail M could be $[+1, |-2, +6, |+5, -3, +4, |-7]$.

Assertion trails

When building a non-recursive implementation the notion of valuation is extended.

Definition (Assertion trail)

Assertion trail is a list of literals, some of which are marked as *decision literals*. Decision literals split the trail into *levels*.

Example

A trail M could be $[+1, |-2, +6, |+5, -3, +4, |-7]$.

Overview

Formal system of Krstić and Goel [KG07]

Decide:
$$\frac{l \in L \qquad l, \bar{l} \notin M}{M := M\ |l}$$

UnitPropagate:
$$\frac{l \vee l_1 \vee \ldots \vee l_k \in F \qquad \bar{l}_1, \ldots, \bar{l}_k \in M \qquad l, \bar{l} \notin M}{M := M\ l}$$

Conflict:
$$\frac{C = no\_cflct \qquad \bar{l}_1 \vee \ldots \vee \bar{l}_k \in F \qquad l_1, \ldots, l_k \in M}{C := \{l_1, \ldots, l_k\}}$$

Explain:
$$\frac{l \in C \qquad l \vee \bar{l}_1 \vee \ldots \vee \bar{l}_k \in F \qquad l_1, \ldots, l_k \prec l}{C := C \cup \{l_1, \ldots, l_k\} \setminus \{l\}}$$

Learn:
$$\frac{C = \{l_1, \ldots, l_k\} \qquad \bar{l}_1 \vee \ldots \vee \bar{l}_k \notin F}{F := F \cup \{\bar{l}_1 \vee \ldots \vee \bar{l}_k\}}$$

Backjump:
$$\frac{C = \{l, l_1, \ldots, l_k\} \qquad \bar{l} \vee \bar{l}_1 \vee \ldots \vee \bar{l}_k \in F \qquad \text{level } l > m \geq \text{level } l_i}{C := no\_cflct \qquad M := M^{[m]}\ \bar{l}}$$

Forget:
$$\frac{C = no\_cflct \qquad c \in F \qquad F \setminus c \vDash c}{F := F \setminus c}$$

Restart:
$$\frac{C = no\_cflct}{M := M^{[0]}}$$

Solver state - (F, M, C)

- F - formula
- M - valuation (trail)
- C - conflict analysis clause

DPLL search - a simplified system

Decide:

$$\frac{l \in F \quad l, \bar{l} \notin M}{M := M \mid l}$$

UnitPropagate:

$$\frac{l \vee l_1 \vee \ldots \vee l_k \in F \qquad \bar{l}_1, \ldots, \bar{l}_k \in M \qquad l, \bar{l} \notin M}{M := M\ l}$$

Backtrack:

$$\frac{M \vDash \neg\, F \quad M = M' \mid l\ M'' \quad \text{decisions}\ M'' = [\,]}{M := M'\ \bar{l}}$$

DPLL search - a simplified system

Decide:

$$\frac{l \in F \quad l, \bar{l} \notin M}{M := M \,|\, l}$$

UnitPropagate:

$$\frac{l \vee l_1 \vee \ldots \vee l_k \in F \qquad \bar{l}_1, \ldots, \bar{l}_k \in M \qquad l, \bar{l} \notin M}{M := M \; l}$$

Backtrack:

$$\frac{M \vDash \neg F \quad M = M' \,|\, l \; M'' \quad \text{decisions } M'' = [\,]}{M := M' \; \bar{l}}$$

DPLL search - a simplified system

Decide:

$$\frac{l \in F \quad l, \bar{l} \notin M}{M := M \mid l}$$

UnitPropagate:

$$\frac{l \vee l_1 \vee \ldots \vee l_k \in F \qquad \bar{l}_1, \ldots, \bar{l}_k \in M \qquad l, \bar{l} \notin M}{M := M\ l}$$

Backtrack:

$$\frac{M \vDash \neg F \quad M = M' \mid l\ M'' \quad \text{decisions } M'' = [\,]}{M := M'\ \bar{l}}$$

DPLL search - a simplified system

Decide:

$$\frac{l \in F \quad l, \bar{l} \notin M}{M := M \,|\, l}$$

UnitPropagate:

$$\frac{l \vee l_1 \vee \ldots \vee l_k \in F \qquad \bar{l}_1, \ldots, \bar{l}_k \in M \qquad l, \bar{l} \notin M}{M := M \; l}$$

Backtrack:

$$\frac{M \vDash \neg\, F \quad M = M' \,|\, l \; M'' \quad \text{decisions } M'' = [\,]}{M := M' \; \bar{l}}$$

Example

$F = [[-1, +2], [-1, -3], [-2, +4, +5], [+3, -4, -5], [-4, +5]]$

Function applied	sat?	M
	UNDEF	$[\,]$
Decide (l = $+1$)	UNDEF	$[\vert +1]$
UnitProp (c = $[-1, +2]$, l = $+2$)	UNDEF	$[\vert +1, +2]$
UnitProp (c = $[-1, -3]$, l = -3)	UNDEF	$[\vert +1, +2, -3]$
Decide (l = $+4$)	UNDEF	$[\vert +1, +2, -3, \vert +4]$
UnitProp (c = $[-4, +5]$, l = $+5$)	UNDEF	$[\vert +1, +2, -3, \vert +4, +5]$
Backtrack ($M \vDash\neg [+3, -4, -5]$)	UNDEF	$[\vert +1, +2, -3, -4]$
UnitProp (c = $[-2, +4, +5]$, l = $+5$)	UNDEF	$[\vert +1, +2, -3, -4, +5]$
$M \nvDash\neg F$, $(\text{vars } M) = (\text{vars } F)$	SAT	$[\vert +1, +2, -3, -4, +5]$

Example

$F = [[-1, +2], [-1, -3], [-2, +4, +5], [+3, -4, -5], [-4, +5]]$

Function applied	sat?	M
	UNDEF	$[\,]$
Decide (l = $+1$)	UNDEF	$[\vert +1]$
UnitProp (c = $[-1, +2]$, l = $+2$)	UNDEF	$[\vert +1, +2]$
UnitProp (c = $[-1, -3]$, l = -3)	UNDEF	$[\vert +1, +2, -3]$
Decide (l = $+4$)	UNDEF	$[\vert +1, +2, -3, \vert +4]$
UnitProp (c = $[-4, +5]$, l = $+5$)	UNDEF	$[\vert +1, +2, -3, \vert +4, +5]$
Backtrack ($M \vDash \neg [+3, -4, -5]$)	UNDEF	$[\vert +1, +2, -3, -4]$
UnitProp (c = $[-2, +4, +5]$, l = $+5$)	UNDEF	$[\vert +1, +2, -3, -4, +5]$
$M \nvDash \neg F$, (vars M) = (vars F)	SAT	$[\vert +1, +2, -3, -4, +5]$

Example

$F = [[-1, +2], [-1, -3], [-2, +4, +5], [+3, -4, -5], [-4, +5]]$

Function applied	sat?	M
	UNDEF	$[\,]$
Decide (l = $+1$)	UNDEF	$[\vert +1]$
UnitProp (c = $[-1, +2]$, l = $+2$)	UNDEF	$[\vert +1, +2]$
UnitProp (c = $[-1, -3]$, l = -3)	UNDEF	$[\vert +1, +2, -3]$
Decide (l = $+4$)	UNDEF	$[\vert +1, +2, -3, \vert +4]$
UnitProp (c = $[-4, +5]$, l = $+5$)	UNDEF	$[\vert +1, +2, -3, \vert +4, +5]$
Backtrack ($M \vDash \neg [+3, -4, -5]$)	UNDEF	$[\vert +1, +2, -3, -4]$
UnitProp (c = $[-2, +4, +5]$, l = $+5$)	UNDEF	$[\vert +1, +2, -3, -4, +5]$
$M \nvDash \neg F$, (vars M) = (vars F)	SAT	$[\vert +1, +2, -3, -4, +5]$

Example

$F = [[-1, +2], [-1, -3], [-2, +4, +5], [+3, -4, -5], [-4, +5]]$

Function applied	sat?	M
	UNDEF	$[\,]$
Decide (l = $+1$)	UNDEF	$[\vert +1]$
UnitProp (c = $[-1, +2]$, l = $+2$)	UNDEF	$[\vert +1, +2]$
UnitProp (c = $[-1, -3]$, l = -3)	UNDEF	$[\vert +1, +2, -3]$
Decide (l = $+4$)	UNDEF	$[\vert +1, +2, -3, \vert +4]$
UnitProp (c = $[-4, +5]$, l = $+5$)	UNDEF	$[\vert +1, +2, -3, \vert +4, +5]$
Backtrack ($M \vDash \neg [+3, -4, -5]$)	UNDEF	$[\vert +1, +2, -3, -4]$
UnitProp (c = $[-2, +4, +5]$, l = $+5$)	UNDEF	$[\vert +1, +2, -3, -4, +5]$
$M \nvDash \neg F$, (vars M) = (vars F)	SAT	$[\vert +1, +2, -3, -4, +5]$

Example

$F = [[-1, +2], [-1, -3], [-2, +4, +5], [+3, -4, -5], [-4, +5]]$

Function applied	sat?	M
	UNDEF	$[\,]$
Decide (l = $+1$)	UNDEF	$[\mid +1]$
UnitProp (c = $[-1, +2]$, l = $+2$)	UNDEF	$[\mid +1, +2]$
UnitProp (c = $[-1, -3]$, l = -3)	UNDEF	$[\mid +1, +2, -3]$
Decide (l = $+4$)	UNDEF	$[\mid +1, +2, -3, \mid +4]$
UnitProp (c = $[-4, +5]$, l = $+5$)	UNDEF	$[\mid +1, +2, -3, \mid +4, +5]$
Backtrack ($M \vDash \neg [+3, -4, -5]$)	UNDEF	$[\mid +1, +2, -3, -4]$
UnitProp (c = $[-2, +4, +5]$, l = $+5$)	UNDEF	$[\mid +1, +2, -3, -4, +5]$
$M \nvDash \neg F$, $(\text{vars } M) = (\text{vars } F)$	SAT	$[\mid +1, +2, -3, -4, +5]$

Example

$F = [[-1, +2], [-1, -3], [-2, +4, +5], [+3, -4, -5], [-4, +5]]$

Function applied	sat?	M
	UNDEF	$[\,]$
Decide (l = $+1$)	UNDEF	$[\vert +1]$
UnitProp (c = $[-1, +2]$, l = $+2$)	UNDEF	$[\vert +1, +2]$
UnitProp (c = $[-1, -3]$, l = -3)	UNDEF	$[\vert +1, +2, -3]$
Decide (l = $+4$)	UNDEF	$[\vert +1, +2, -3, \vert +4]$
UnitProp (c = $[-4, +5]$, l = $+5$)	UNDEF	$[\vert +1, +2, -3, \vert +4, +5]$
Backtrack ($M \vDash \neg [+3, -4, -5]$)	UNDEF	$[\vert +1, +2, -3, -4]$
UnitProp (c = $[-2, +4, +5]$, l = $+5$)	UNDEF	$[\vert +1, +2, -3, -4, +5]$
$M \nvDash \neg F$, (vars M) = (vars F)	SAT	$[\vert +1, +2, -3, -4, +5]$

Example

$F = [[-1,+2],[-1,-3],[-2,+4,+5],[+3,-4,-5],[-4,+5]]$

Function applied	sat?	M
	UNDEF	$[\,]$
Decide (l = $+1$)	UNDEF	$[\vert +1]$
UnitProp (c = $[-1,+2]$, l = $+2$)	UNDEF	$[\vert +1,+2]$
UnitProp (c = $[-1,-3]$, l = -3)	UNDEF	$[\vert +1,+2,-3]$
Decide (l = $+4$)	UNDEF	$[\vert +1,+2,-3,\vert +4]$
UnitProp (c = $[-4,+5]$, l = $+5$)	UNDEF	$[\vert +1,+2,-3,\vert +4,+5]$
Backtrack ($M \vDash \neg [+3,-4,-5]$)	UNDEF	$[\vert +1,+2,-3,-4]$
UnitProp (c = $[-2,+4,+5]$, l = $+5$)	UNDEF	$[\vert +1,+2,-3,-4,+5]$
$M \nvDash \neg F$, (vars M) = (vars F)	SAT	$[\vert +1,+2,-3,-4,+5]$

Example

$F = [[-1, +2], [-1, -3], [-2, +4, +5], [+3, -4, -5], [-4, +5]]$

Function applied	sat?	M
	UNDEF	$[\,]$
Decide (l = $+1$)	UNDEF	$[\vert +1]$
UnitProp (c = $[-1, +2]$, l = $+2$)	UNDEF	$[\vert +1, +2]$
UnitProp (c = $[-1, -3]$, l = -3)	UNDEF	$[\vert +1, +2, -3]$
Decide (l = $+4$)	UNDEF	$[\vert +1, +2, -3, \vert +4]$
UnitProp (c = $[-4, +5]$, l = $+5$)	UNDEF	$[\vert +1, +2, -3, \vert +4, +5]$
Backtrack ($M \vDash \neg [+3, -4, -5]$)	UNDEF	$[\vert +1, +2, -3, -4]$
UnitProp (c = $[-2, +4, +5]$, l = $+5$)	UNDEF	$[\vert +1, +2, -3, -4, +5]$
$M \nvDash \neg F$, $(\text{vars } M) = (\text{vars } F)$	SAT	$[\vert +1, +2, -3, -4, +5]$

Example

$F = [[-1, +2], [-1, -3], [-2, +4, +5], [+3, -4, -5], [-4, +5]]$

Function applied	sat?	M
	UNDEF	$[\,]$
Decide (l = $+1$)	UNDEF	$[\mid +1]$
UnitProp (c = $[-1, +2]$, l = $+2$)	UNDEF	$[\mid +1, +2]$
UnitProp (c = $[-1, -3]$, l = -3)	UNDEF	$[\mid +1, +2, -3]$
Decide (l = $+4$)	UNDEF	$[\mid +1, +2, -3, \mid +4]$
UnitProp (c = $[-4, +5]$, l = $+5$)	UNDEF	$[\mid +1, +2, -3, \mid +4, +5]$
Backtrack ($M \vDash\neg [+3, -4, -5]$)	UNDEF	$[\mid +1, +2, -3, -4]$
UnitProp (c = $[-2, +4, +5]$, l = $+5$)	UNDEF	$[\mid +1, +2, -3, -4, +5]$
$M \nvDash\neg F$, $(\text{vars } M) = (\text{vars } F)$	SAT	$[\mid +1, +2, -3, -4, +5]$

Transition relation - formal definition

Definition (State)

State (M, F) is an ordered pair of an assertion trail M and formula F.

Definition

decide $(M_1, F_1)\ (M_2, F_2) \iff$

$$\exists l. \quad \text{var } l \in \text{vars } F_1 \;\wedge\; l \notin M_1 \;\wedge\; \bar{l} \notin M_1 \;\wedge\; M_2 = M_1 \,@\, l^{\top} \;\wedge\; F_2 = F_1$$

backtrack $(M_1, F_1)\ (M_2, F_2) \iff$

$$M_1 \vDash\neg\, F_1 \;\wedge\; \text{decisions } M_1 \neq [\,] \;\wedge\; M_2 = \text{prefixBeforeLastDecision } M_1 \,@\, \overline{\text{lastDecision } M_1}^{\perp} \;\wedge\; F_2 = F_1$$

Transition relation - formal definition

Definition

$$(M_1, F_1) \to (M_2, F_2) \iff \begin{array}{l} \text{decide } (M_1, F_1)\ (M_2, F_2)\ \vee \\ \text{backtrack } (M_1, F_1)\ (M_2, F_2)\ \vee \\ \text{unitPropagate } (M_1, F_1)\ (M_2, F_2) \end{array}$$

The relation $\to^*$ is the *transitive and reflexive closure* of the $\to$ relation.

The state (M, F) is a final state if it is minimal wrt. the relation $\to$, i.e., if there is no state (M', F') st. $(M, F) \to (M', F')$.

Theorem (Soundness)

Let $([\,], F_0) \rightarrow^* (M, F)$.

- *If*
  1. *no conflict (i.e.,* $M \not\models \neg\, F$*),*
  2. *the rule Decide is not applicable (i.e.,* $\mathrm{var}\ l \in \mathrm{vars}\ F_0$, $l \notin M$ *and* $\bar{l} \notin M$*)*

  then F_0 *is satisfiable and* M *is its model (i.e.,* $\mathrm{sat}\ F_0$ *and* $\mathrm{model}\ M\ F_0$*).*
- *If*
  1. *conflict (i.e.,* $M \models \neg\, F$*),*
  2. *the rule Backtrack is not applicable (i.e.,* $(\mathrm{decisions}\ M) = [\,]$*),*

  then F_0 *is unsatisfiable (i.e.,* $\neg \mathrm{sat}\ F_0$*).*

Theorem (Soundness)

Let $([\,], F_0) \rightarrow^* (M, F)$.

- *If*
  1. *no conflict (i.e.,* $M \not\models \neg\, F$*),*
  2. *the rule Decide is not applicable (i.e.,* $\text{var}\ l \in \text{vars}\ F_0,\ l \notin M$ *and* $\bar{l} \notin M$*)*

  then F_0 *is satisfiable and* M *is its model (i.e.,* $\text{sat}\ F_0$ *and* $\text{model}\ M\ F_0$*).*
- *If*
  1. *conflict (i.e.,* $M \models \neg\, F$*),*
  2. *the rule Backtrack is not applicable (i.e.,* $(\text{decisions}\ M) = [\,]$*),*

  then F_0 *is unsatisfiable (i.e.,* $\neg\text{sat}\ F_0$*).*

Theorem (Pre-Completeness)

In every finite state (M, F) one of the following holds:

1. *the rule Backtrack is not applicable (i.e., $M \vDash \neg\, F$ and decisions $M = [\,]$)*
2. *the rule Decide is not applicable (i.e., $M \nvDash \neg\, F$ and var $l \in$ vars F_0, $l \notin M$ and $\bar{l} \notin M$)*

Theorem (Termination)

Relation $\rightarrow$ is well-founded, i.e., there is no infinite descending chain

$$([\,], F_0) \rightarrow (M_1, F_1) \rightarrow (M_2, F_2) \rightarrow \ldots$$

How are these theorems proved?

Invariants

$Invariant_{consistent}$:	consistent M
$Invariant_{distinct}$:	distinct M
$Invariant_{varsM}$:	vars $M \subseteq$ vars F
$Invariant_{impliedLiterals}$:	$\forall l.\ l \in M \implies (F \ @ \ \text{decisionsTo}\ l\ M) \vDash l$

Theorem

If $([\,], F_0) \rightarrow^* (M, F)$, *then all invariants hold in the state* (M, F).

How are these theorems proved?

The termination is proved using well-founded orderings.

Definition

$$l_1 \prec^{lit} l_2 \iff (\text{isDecision } l_1) \wedge \neg(\text{isDecision } l_2)$$

Definition

$$M_1 \succ_M M_2 \iff M_1 \prec^{lit}_{lex} M_2,$$

where $\prec^{lit}_{lex}$ is a lexicographic extension of relation $\prec^{lit}$.

Definition

$$\begin{aligned} M_1 \succ^r_M M_2 \quad \iff \quad & (\text{consistent } M_1) \wedge (\text{distinct } M_1) \wedge (\text{vars } M_1) \subseteq Vbl \\ & (\text{consistent } M_2) \wedge (\text{distinct } M_2) \wedge (\text{vars } M_2) \subseteq Vbl \\ & M_1 \succ_M M_2 \end{aligned}$$

Overview

Shallow embedding into HOL - characteristics

- Program is expressed as a set of recursive HOL functions.
- Proof methods are just standard induction principles and equational reasoning.
- No specialized program logic (e.g., Hoare logic) is necessary.
- Executable code can be automatically generated.
- Side-effects are impossible.

Shallow embedding into HOL - characteristics

- Program is expressed as a set of recursive HOL functions.
- Proof methods are just standard induction principles and equational reasoning.
- No specialized program logic (e.g., Hoare logic) is necessary.
- Executable code can be automatically generated.
- Side-effects are impossible.

Shallow embedding into HOL - characteristics

- Program is expressed as a set of recursive HOL functions.
- Proof methods are just standard induction principles and equational reasoning.
- No specialized program logic (e.g., Hoare logic) is necessary.
- Executable code can be automatically generated.
- Side-effects are impossible.

Shallow embedding into HOL - characteristics

- Program is expressed as a set of recursive HOL functions.
- Proof methods are just standard induction principles and equational reasoning.
- No specialized program logic (e.g., Hoare logic) is necessary.
- Executable code can be automatically generated.
- Side-effects are impossible.

Shallow embedding into HOL - characteristics

- Program is expressed as a set of recursive HOL functions.
- Proof methods are just standard induction principles and equational reasoning.
- No specialized program logic (e.g., Hoare logic) is necessary.
- Executable code can be automatically generated.
- Side-effects are impossible.

Embedding of a SAT solver

- Embedding of the classic DPLL was an easy to do excercise [MJ09].
- Embedding of a modern SAT solver was a big challenge [Mar09].

Embedding of a SAT solver

- Embedding of the classic DPLL was an easy to do excercise [MJ09].
- Embedding of a modern SAT solver was a big challenge [Mar09].

Embedding of a modern MiniSat-like SAT solver

Positive:

- Implementation follows state transition systems.
- All algorithms described by state stransition systems are implemented.
- Also, major low-level implementation techniques are implemented.

Negative:

- There are no in-place modifications of data.
- Global variables are gruped to form a solver state which is explicitly passed around function calls.

Embedding of a modern MiniSat-like SAT solver

Positive:

- Implementation follows state transition systems.
- All algorithms described by state stransition systems are implemented.
- Also, major low-level implementation techniques are implemented.

Negative:

- There are no in-place modifications of data.
- Global variables are gruped to form a solver state which is explicitly passed around function calls.

Embedding of a modern MiniSat-like SAT solver

Positive:

- Implementation follows state transition systems.
- All algorithms described by state stransition systems are implemented.
- Also, major low-level implementation techniques are implemented.

Negative:

- There are no in-place modifications of data.
- Global variables are gruped to form a solver state which is explicitly passed around function calls.

Embedding of a modern MiniSat-like SAT solver

Positive:

- Implementation follows state transition systems.
- All algorithms described by state stransition systems are implemented.
- Also, major low-level implementation techniques are implemented.

Negative:

- There are no in-place modifications of data.
- Global variables are gruped to form a solver state which is explicitly passed around function calls.

Embedding of a modern MiniSat-like SAT solver

Positive:

- Implementation follows state transition systems.
- All algorithms described by state stransition systems are implemented.
- Also, major low-level implementation techniques are implemented.

Negative:

- There are no in-place modifications of data.
- Global variables are gruped to form a solver state which is explicitly passed around function calls.

Solver state

```
record State =
   "getM"                :: LiteralTrail
   "getF"                :: Formula
   "getSATFlag"          :: ExtendedBool
   "getConflictFlag"     :: bool
   "getConflictClause"   :: pClause
   "getQ"                :: "Literal list"
   "getReason"           :: "Literal ⇒ pClause option"
   "getWatch1"           :: "pClause ⇒ Literal option"
   "getWatch2"           :: "pClause ⇒ Literal option"
   "getWatchList"        :: "Literal ⇒ pClause list"
   "getC"                :: Clause
   "getCl"               :: Literal
   "getCll"              :: Literal
```

Solver state

```
record State =
   "getM"                 :: LiteralTrail
   "getF"                 :: Formula
   "getSATFlag"           :: ExtendedBool
   "getConflictFlag"      :: bool
   "getConflictClause"    :: pClause
   "getQ"                 :: "Literal list"
   "getReason"            :: "Literal ⇒ pClause option"
   "getWatch1"            :: "pClause ⇒ Literal option"
   "getWatch2"            :: "pClause ⇒ Literal option"
   "getWatchList"         :: "Literal ⇒ pClause list"
   "getC"                 :: Clause
   "getCl"                :: Literal
   "getCll"               :: Literal
```

applyDecide – implements Decide rule

definition applyDecide :: "State ⇒ Variable set ⇒ State"
where

```
"applyDecide state decisionVars =
   assertLiteral (selectLiteral state decisionVars) True state
"
```

The main solver function

```
definition solve ::  "Formula ⇒ ExtendedBool"
where
"solve F0 = getSATFlag (solve_loop (initialize F0 initialState) (vars F0))"
definition solve_loop_body ::  "State ⇒ Variable set ⇒ State"
where
"solve_loop_body state decisionVars =
  (let state_up = exhaustiveUnitPropagate state in
  (if (getConflictFlag state_up) then
    (if (currentLevel (getM state_up)) = 0 then
      state_up⦇ getSATFlag := False ⦈
    else
      let state_c = applyConflict state_up in
      let state_e = applyExplainUIP state_c in
      let state_l = applyLearn state_e in
      let state_b = applyBackjump state_l in
      state_b
    )
  else
    (if (vars (elements (getM state_up)) ⊇ decisionVars) then
      state_up⦇ getSATFlag := TRUE ⦈
    else
      applyDecide state_up decisionVars
    )
  ))

"
```

solve_loop – a total recursive function

```
function (domintros, tailrec) solve_loop ::
     "State ⇒ Variable set ⇒ State"
where
"solve_loop state decisionVars =
   (if (getSATFlag state) ≠ UNDEF then
      state
   else
      let state' = solve_loop_body state decisionVars in
      solve_loop state' decisionVars
   )
"
by pat_completeness auto
```

Two-watch literal scheme – the most complex function

```
primrec
notifyWatches_loop :: "Literal ⇒ pClause list ⇒ pClause list ⇒ State ⇒ State"
where
"notifyWatches_loop literal [] newWl state =
  state(| getWatchList := (getWatchList state)(literal := newWl) |)" |
"notifyWatches_loop literal (clause # list') newWl state =
  (let state' = (if Some literal = (getWatch1 state clause) then (swapWatches clause state)
                 else state) in
  case (getWatch1 state' clause) of Some w1 ⇒ (
  case (getWatch2 state' clause) of Some w2 ⇒ (
  (if (literalTrue w1 (elements (getM state'))) then
   notifyWatches_loop literal list' (newWl @ [clause]) state'
  else
    (case (getNonWatchedUnfalsifiedLiteral ((getF state') ! clause) w1 w2 (getM state')) of
    Some l' ⇒
      notifyWatches_loop literal list' newWl (setWatch2 clause l' state') |
    None ⇒
      (if (literalFalse w1 (elements (getM state'))) then
        let state'' = state'(| getConflictFlag := True, getConflictClause := clause |) in
        notifyWatches_loop literal list' (newWl @ [clause]) state''
      else
        let state'' =
          state'(| getQ := (if w1 el (getQ state') then (getQ state') else (getQ state') @ [w1])|) in
        let state''' = (setReason w1 clause state'') in
        notifyWatches_loop literal list' (newWl @ [clause]) state''')))))))"
```

Total correctness

Theorem

$$\text{solve } F_0 = SAT \iff \text{sat } F_0$$

How is it proved?

- Correctness proofs for state transition systems were reused.
- New invariants (24 totally) were introduced.

A complex invariant

$$\begin{aligned}&\forall c.\ c < |F| \implies M \vDash\neg (watch_1\ c) \implies \\ &\quad (\exists l.\ l \in c\ \wedge\ M \vDash l\ \wedge\ \text{level}\ l \leq \text{level}\ \overline{(watch_1\ c)}) \vee \\ &\quad (\forall l.\ l \in c\ \wedge\ l \neq (watch_1\ c)\ \wedge\ l \neq (watch_2\ c) \implies \\ &\qquad M \vDash\neg l\ \wedge\ \text{level}\ \bar{l} \leq \text{level}\ \overline{(watch_1\ c)}).\end{aligned}$$

- Partial termination of recursive functions was proved, using the same orderings as for state transition systems.

How is it proved?

- Correctness proofs for state transition systems were reused.
- New invariants (24 totally) were introduced.

A complex invariant

$$\forall c.\ c < |F| \implies M \vDash \neg (watch_1\ c) \implies$$
$$(\exists l.\ l \in c \ \wedge\ M \vDash l\ \wedge\ \text{level}\ l \leq \text{level}\ \overline{(watch_1\ c)}) \vee$$
$$(\forall l.\ l \in c\ \wedge\ l \neq (watch_1\ c)\ \wedge\ l \neq (watch_2\ c) \implies$$
$$M \vDash \neg l\ \wedge\ \text{level}\ \overline{l} \leq \text{level}\ \overline{(watch_1\ c)}).$$

- Partial termination of recursive functions was proved, using the same orderings as for state transition systems.

How is it proved?

- Correctness proofs for state transition systems were reused.
- New invariants (24 totally) were introduced.

A complex invariant

$$\begin{aligned}
&\forall c.\ c < |F| \implies M \vDash \neg\, (watch_1\ c) \implies \\
&\quad (\exists l.\ l \in c \ \wedge\ M \vDash l \ \wedge\ \text{level } l \leq \text{level } \overline{(watch_1\ c)}) \vee \\
&\quad (\forall l.\ l \in c \ \wedge\ l \neq (watch_1\ c) \ \wedge\ l \neq (watch_2\ c) \implies \\
&\qquad M \vDash \neg\, l \ \wedge\ \text{level } \bar{l} \leq \text{level } \overline{(watch_1\ c)}).
\end{aligned}$$

- Partial termination of recursive functions was proved, using the same orderings as for state transition systems.

How is it proved?

- Correctness proofs for state transition systems were reused.
- New invariants (24 totally) were introduced.

A complex invariant

$$\begin{array}{l} \forall c.\ c < |F| \implies M \vDash\neg (watch_1\ c) \implies \\ \quad (\exists l.\ l \in c \ \wedge\ M \vDash l \ \wedge\ \text{level } l \leq \text{level } \overline{(watch_1\ c)}) \vee \\ \quad (\forall l.\ l \in c \ \wedge\ l \neq (watch_1\ c) \ \wedge\ l \neq (watch_2\ c) \implies \\ \qquad M \vDash\neg l \ \wedge\ \text{level } \overline{l} \leq \text{level } \overline{(watch_1\ c)}). \end{array}$$

- Partial termination of recursive functions was proved, using the same orderings as for state transition systems.

Some numbers

- Around 1 man-year effort.
- $\approx$ 25 000 lines of Isar code.
- Generated PDF-s $\approx$ 700 pages.

Further work

- Extract executable code from specifications.
- Use monadic programming to get imperative features

Non-monadic programming

```
definition setWatch1 :: "pClause ⇒ Literal ⇒ State ⇒ State"
where
"setWatch1 clause literal state =
  let state' = state⦇ getWatch1 := (getWatch1 state)(clause := Some literal) ⦈ in

  addToWatchList literal clause state' "
```

Monadic programming

```
definition setWatch1 :: "nat ⇒ Literal ⇒ unit StateTransformer"
where
"setWatch1 clause literal =
  do
    updateWatch1 clause (Some literal);
    addToWatchList literal clause

  done"
```

Further work

- Extract executable code from specifications.
- Use monadic programming to get imperative features

Non-monadic programming

```
definition setWatch1 ::  "pClause ⇒ Literal ⇒ State ⇒ State"
where
"setWatch1 clause literal state =
  let state' = state⦇ getWatch1 := (getWatch1 state)(clause := Some literal) ⦈ in

  addToWatchList literal clause state' "
```

Monadic programming

```
definition setWatch1 ::  "nat ⇒ Literal ⇒ unit StateTransformer"
where
"setWatch1 clause literal =
  do
    updateWatch1 clause (Some literal);
    addToWatchList literal clause

  done"
```

Further work

- Extract executable code from specifications.
- Use monadic programming to get imperative features

Non-monadic programming

```
definition setWatch1 ::  "pClause ⇒ Literal ⇒ State ⇒ State"
where
"setWatch1 clause literal state =
   let state' = state⦇ getWatch1 := (getWatch1 state)(clause := Some literal) ⦈ in

   addToWatchList literal clause state' "
```

Monadic programming

```
definition setWatch1 ::  "nat ⇒ Literal ⇒ unit StateTransformer"
where
"setWatch1 clause literal =
   do
      updateWatch1 clause (Some literal);
      addToWatchList literal clause

   done"
```

Further work

- Extract executable code from specifications.
- Use monadic programming to get imperative features

Non-monadic programming

```
definition setWatch1 ::  "pClause ⇒ Literal ⇒ State ⇒ State"
where
"setWatch1 clause literal state =
  let state' = state⦇ getWatch1 := (getWatch1 state)(clause := Some literal) ⦈ in

  addToWatchList literal clause state' "
```

Monadic programming

```
definition setWatch1 ::  "nat ⇒ Literal ⇒ unit StateTransformer"
where
"setWatch1 clause literal =
  do
    updateWatch1 clause (Some literal);
    addToWatchList literal clause

  done"
```

S. Krstic and A. Goel.
Architecting solvers for SAT modulo theories: Nelson-oppen with DPLL.
In *FroCos*, pp. 1–27, Liverpool, 2007.

Filip Marić.
Formalization and implementation of SAT solvers.
Journal of Automated Reasoning, 2009.

Filip Marić, Predrag Janičić.
Formal Correctness Proof for DPLL Procedure.
Informatica, 2009.

Filip Marić.
A formally verified SAT solver.
submitted to *TCS*.