Kernel of Truth based SAT proof checker

Dan Andrei Marian, Ecole Polytechnique

SRI International

August 27th, 2010

Outline

Introduction

SAT Solver

- Program to determine whether a logic formula is satisfiable.

Examples

- Unsat formula $(a \bigvee b) \bigwedge (\neg a \bigvee b) \bigwedge (a \bigvee \neg b) \bigwedge (\neg a \bigvee \neg b)$
- Sat formula $(a \bigvee b \bigvee c) \bigwedge (\neg a \bigvee \neg b \bigvee \neg c)$
- Standard input formulas are in CNF

Output

- Usually, the SAT solver answers sat/unsat.
- Critical applications need guarantees that the answer is correct.
- Example: Modern SAT solvers generate proofs to justify their answer

Proof format

When the logic formula is satisfiable, the solver can return the assignment of all the variables

Example: $a = T, b = F, c = T$ makes the formula $(a \bigvee b \bigvee c) \bigwedge(\neg a \bigvee \neg b \bigvee \neg c)$ true

Problem: how to prove that a formula is unsatisfiable?

- Using the resolution rule:

$$\frac{A \bigvee A_1 \bigvee \ldots \bigvee A_n \quad \neg A \bigvee B_1 \bigvee \ldots \bigvee B_m}{A_1 \bigvee \ldots \bigvee A_n \bigvee B_1 \bigvee \ldots B_m} \tag{1}$$

- If the two hypothesis clauses are true, then the resolvent is also true

Approach: Using the initial clauses, try to obtain the empty clause (contradiction)

A small example

$$(a \bigvee b) \bigwedge (\neg a \bigvee b) \bigwedge (a \bigvee \neg b) \bigwedge (\neg a \bigvee \neg b)$$

We resolve the first two clauses, and we obtain: (b)

The last two clauses give the resolvent: $(\neg b)$

By resolving the two previous derived clauses, we obtain the empty clause

Proof format idea: Indicate the path (order in which the clauses must be resolved) to the empty clause

DIMACS format

The previous formula is represented in CNF DIMACS format:

```
p cnf 2 4
1 2 0
-1 2 0
1 -2 0
-1 -2 0
```

The proof generated by a SAT solver is:

```
0 1 2 s s
1 -1 2 s s
2 1 -2 s s
3 -1 -2 s s
4 * 1 3 s //(-1)
5 * 4 2 s //(-2)
6 * 4 0 5 s //(empty_clause)
```

Who checks the checker?

The SAT solver produces a proof that is verified by the checker

- Can we trust the checker?

Kernel of Truth approach

- We build a small set of FOL rules in PVS
- Proofs can be produced using these inference rules
- We build a verified checker for the SAT proofs based on KoT
- The best way to find out if you can trust somebody is to trust them. (Ernest Hemingway)

Overview of PVS

Prototype Verification System

It is an environment for specification and proving developped at SRI.

Advantages:

- Subtle errors are revealed by trying to prove the properties
- Testing models is possible by generating Lisp code from PVS.

Example:

```
list [T: TYPE]: DATATYPE
BEGIN
 null: null?
 cons (car: T, cdr:list):cons?
END list
```

Presenting KoT - Terms and Formulas

```
term  : DATATYPE
   BEGIN
   v(v_index: nat): var?
   apply(fun: (fun?),
         args: {ss: list[term] |
  length(ss) = arity(fun)}): apply?
  END term
fmla: DATATYPE
  BEGIN
   atom(pred: (pred?), args: {ss: list[term] |
      length(ss) = arity(pred)}): atom?
   f_not(arg: fmla): f_not?
   f_or(arg1, arg2: fmla): f_or?
   f_exists(bvar: (var?), body: fmla): f_exists?
  END fmla
```

Presenting the Kernel of Truth

Inference rules

- Using one sided sequents:
  - $$\frac{}{\vdash A, \neg A, \ldots}$$
  - $$\frac{\vdash A_1, \ldots, A_n}{\vdash B_1, \ldots B_m}, A_1, \ldots, A_n \subset B_1, \ldots B_m$$
  - $$\frac{\vdash A, B}{\vdash A \bigvee B}$$
  - $$\frac{\vdash \neg A \quad \vdash \neg B}{\vdash \neg(A \bigvee B)}$$
  - $$\ldots$$

KoT proofs

Tree proof for resolution

- $$\dfrac{\dfrac{\dfrac{axiom}{\vdash \neg p, \neg\neg p} \qquad \dfrac{axiom}{\vdash \neg\Delta, \Delta}}{\vdash \neg p, \neg(\neg p \bigvee \Delta), \Delta} \qquad \dfrac{axiom}{\vdash \neg\Gamma, \Gamma}}{\dfrac{\vdash \neg(p \bigvee \Gamma), \neg(\neg p \bigvee \Delta), \Gamma, \Delta}{\vdash \neg(p \bigvee \Gamma), \neg(\neg p \bigvee \Delta), \Gamma \bigvee \Delta}}$$

Building the SAT proof checker

The main function of the checker

-
```
resolution(nck, ncl) : (tr_clause?) =
    IF tr_clause_true?(nck) THEN ncl
    ELSIF tr_clause_true?(ncl) THEN nck
    ELSE
     LET merged = merge(nck, ncl) IN
     IF exist_pivot?(nck, ncl) THEN
       LET pivot= find_pivot(nck, ncl) IN
       delete_pivot(merged, pivot)
     ELSE
       merged
     ENDIF
    ENDIF
```

SAT proof checker details

PVS extensions in Lisp using the PVSio library for I/O operations

Lisp programs to make small changes to the proof format

Verifying the checker

We prove that for each valid SAT proof exists an equivalent KoT proof

```
o th: THEOREM
         conclusion(proof_th(ntcA, ntcB)) =
   append(
    not_or_reduction(translate_clause(ntcA)),
    append(
        not_or_reduction(translate_clause(ntcB)),
      translate_clause(resolution(ntcA, ntcB))))
  AND checkProof(empty_seq)(proof_th(ntcA, ntcB))
```

Verifying the checker

We extend the proof from one step resolution to chain resolution

- ```
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))
```
```

Some limitations of this solution

The generated code is slow

- Some functions are not tail recursive
- A compromise between efficiency of the code and the ease to prove the required properties
- Expensive `map` operations

Solutions

- Rewrite functions to tail recursive form.
- Write efficient functions and prove the equivalence with previous versions

Tests

Critical operation: the resolution step

Size of the trace (KB)	Time to check (s)
1.2	0.003
1.6	0.015
76	4.7
449	20
963	74
1800	132
15300	1006
37800	1863

Yices SAT solver

SMT Solver developed at SRI

Task: instrument the SAT solver to generate proofs

Achived: generate a valid trace, containing all the initial and learned clauses

Improve: reduce proof size by eliminating unused clauses

Conclusion

So far:

- Present the KoT structure
- Develop the trace checker in PVS
- Verify the checker by proving the equivalence theorems
- Assure compatibility with traces generated by solvers
- Generate proofs for the Yices SAT solver

Developments:

- Optimize current implementation
- Extend instrumentation and verification to rewrite tools and SMT solvers