

G22.3033.11 — Logic and Verification  
Lecture 14

# Review

---

## Last time

- Congruence Closure
- Shostak's Method
- Putting it all Together: CVC

# Outline

---

- Applying SAT techniques in CVC
- Proof-production in CVC
- CVC demo

Sources:

C. Barrett, D. Dill, A. Stump. *Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT*. CAV 2002

C. Barrett, S. Berezin, D. Dill. *A Proof-Producing Boolean Search Engine*.  
Submitted to PDPAR '03.

## Checking the Satisfiability of Arbitrary Formulas

We have discussed a few different methods for checking satisfiability of quantifier-free first-order formulas:

- Nelson-Oppen method
- Congruence closure
- Shostak's method

*All* of these methods assume that the formula to be checked is a *conjunction of literals*.

What if the formula is not a conjunction of literals?

One approach would be to use propositional transformations (such as distributivity and DeMorgan's laws) to transform the formula into disjunctive normal form (DNF) and then test each disjunct separately.

However, this can result in an exponential blow-up in the size of the formula and is thus too costly in practice.

## Checking the Satisfiability of Arbitrary Formulas

Another approach is to look for a consistent assignment of truth values to the literals which makes the formula true.

Note that this is really the combination of two problems:

- A Boolean satisfiability problem to find an assignment of truth values to the Boolean structure of the formula.
- A non-Boolean satisfiability problem to check whether the assignment of truth values to literals is consistent.

We can use SAT-based techniques to solve the first problem, and Nelson-Oppen based techniques to solve the second.

In the remainder of this lecture, we assume that  $Sat_{FO}$  is an algorithm for determining the satisfiability of a conjunction of first-order literals.

## Checking the Satisfiability of Arbitrary Formulas

SVC (the predecessor to CVC) used a simple DPLL recursive search to solve the Boolean satisfiability portion of the problem.

CheckSat( $\Gamma, \phi$ )

IF  $\neg \text{Sat}_{FO}(\Gamma)$  THEN RETURN  $\emptyset$ ;

$\phi' := \text{Simplify}(\Gamma, \phi)$ ;

IF  $\phi' = \text{FALSE}$  THEN RETURN  $\emptyset$ ;

IF  $\phi' = \text{TRUE}$  THEN RETURN  $\Gamma$ ;

$\alpha := \text{FindLiteral}(\phi')$ ;

$\Gamma' := \text{CheckSat}(\Gamma \cup \{\alpha\}, \phi')$ ;

IF  $\Gamma' \neq \emptyset$  THEN RETURN  $\Gamma'$ ;

RETURN  $\text{CheckSat}(\Gamma \cup \{\neg\alpha\}, \phi')$ ;

## Checking the Satisfiability of Arbitrary Formulas

The simple DPLL recursive search works well on small examples.

However, for large examples, it can be very slow.

The performance is highly dependent on the order in which literals are picked.

Also, the DPLL algorithm does not take advantage of recent advances in SAT research.

For these reasons, CVC takes a different approach to solving the Boolean part of the problem.

Basically the approach taken by CVC is to use an enhanced SAT solver which works jointly with the first-order solver to try to find a consistent satisfying set of literals.

## Computing a Propositional Abstraction

Given a quantifier-free first-order formula  $\phi$ , the first step is to compute a *propositional abstraction*  $Abs(\phi)$  of  $\phi$ .

We do this by replacing each atomic formula  $\alpha$  in  $\phi$  by a propositional variable  $p_\alpha$ .

The result is a propositional formula  $Abs(\phi)$  which has the following property:

If  $Abs(\phi)$  is unsatisfiable, then  $\phi$  is unsatisfiable.

However, the converse is not true. It may be the case that  $Abs(\phi)$  is satisfiable but  $\phi$  is not (Why?).

## A Naive Algorithm

Suppose we wish to check the satisfiability of a quantifier-free first-order formula  $\phi$ .

- We first form the propositional abstraction  $Abs(\phi)$ .
- We check  $Abs(\phi)$  for satisfiability using a Boolean SAT solver.
- If  $Abs(\phi)$  is unsatisfiable,  $\phi$  is unsatisfiable.
- Otherwise, let  $\psi$  be a variable assignment satisfying  $Abs(\phi)$ .
- Let  $Abs^{-1}(\psi)$  be the conjunction of first-order literals corresponding to  $\psi$ .
- If  $Sat_{FO}(Abs^{-1}(\psi))$ , then  $\phi$  is satisfiable.
- Otherwise, we *refine*  $Abs(\phi)$  by adding  $\neg\psi$  and repeat.

Since there are only a finite number of possible variable assignments to  $Abs(\phi)$ , the algorithm will eventually terminate.

## Problems with the Naive Approach

Although simple and elegant, the approach just described is not practical.

The following is a list of the main issues that have to be addressed in order to make the algorithm efficient in practice.

- Redundant clauses
- Lazy notification
- Decision heuristics
- Sat heuristics and completeness
- Non-convexity issues

## Redundant Clauses

The main difficulty with the naive approach is that it tends to produce an enormous amount of redundant clauses.

Suppose that the Boolean abstraction  $Abs(\phi)$  contains  $n + 2$  propositional variables.

When a Boolean assignment is returned by the SAT solver, all  $n + 2$  variables will have an assignment.

But what if only 2 of these assignments are sufficient to result in an inconsistency in the atomic formulas associated with the variables?

For each assignment of values to the other  $n$  propositional variables which leads to a satisfying solution, the refinement loop will have to add another clause.

In the worst case,  $2^n$  clauses will be added when a single clause containing the two offending variables would have sufficed.

## Redundant Clauses

To avoid these kinds of redundant clauses, the refinement must be more precise.

In particular, when  $Sat_{FO}$  is given a set of literals to check for consistency, an effort must be made to find the *smallest* possible subset of the given set which is inconsistent.

The refinement should then add a clause derived from *only these* literals.

One way to implement this is to start removing literals from the set and repeatedly call  $Sat_{FO}$  until a minimal inconsistent set is found.

However, this is much too slow to be practical.

A better, but more difficult way to implement this is to instrument  $Sat_{FO}$  so that it is *explicating*, meaning that it keeps track of which facts are used to derive an inconsistency.

This is the approach used in CVC. We will discuss it more in the second half of this lecture.

## Lazy Notification

The naive algorithm is *lazy* in the sense that the SAT solver is used as a black box and  $Sat_{FO}$  is not invoked until a complete solution is obtained.

In contrast, an *eager* approach is to notify  $Sat_{FO}$  incrementally of every decision made by the SAT solver.

Experimental results show that the eager approach is significantly better.

Eager notification requires that  $Sat_{FO}$  be *online*: able quickly to determine the consistency of incrementally more or fewer literals.

Eager notification also requires that the SAT solver be instrumented to inform  $Sat_{FO}$  every time it assigns a variable.

## Naive, Lazy, and Eager Implementations

Example	Naive		Lazy		Eager
	Iterations	Time (s)	Iterations	Time (s)	Time (s)
read0	77	0.14	17	0.09	0.07
pp-pc-s2i	?	> 10000	82	1.36	0.10
pp-invariant	?	> 10000	239	5.81	0.22
v-dlx-pc	?	> 10000	6158	792	3.22
v-dlx-dmem	?	> 10000	?	> 10000	4.12

## Decision Heuristics

SAT solvers like Chaff have sophisticated heuristics for determining which variable to split on.

However, for some first-order examples, the *structure* of the original formula is an important consideration when determining which literal to split on.

For example, CVC includes the **ite** (if-then-else) construct.

Suppose an **ite** expression of the form **ite**( $\alpha, t_1, t_2$ ) appears in the formula being checked.

If  $\alpha$  is set to *true*, then all of the literals in  $t_2$  can be ignored since they no longer affect the formula.

Unfortunately, the SAT solver doesn't know this and as a result, it can waste a lot of time choosing irrelevant variables.

We found that for such examples, it was better to use a depth-first traversal of the original formula to choose splitters than the built-in SAT heuristic.

Again, this requires tighter integration and communication between the two solvers.

## Variable Selection Results

Example	SAT		DFS	
	Decisions	Time (s)	Decisions	Time (s)
bool-dlx1-c	1309	0.69	2522	1.14
bool-dlx2-aa	4974	2.36	792	0.81
bool-dlx2-cc-bug01	10903	11.4	573387	833
v-dlx-pc	4387	3.22	6137	6.10
v-dlx-dmem	5221	4.12	2184	3.48
v-dlx-regfile	6802	5.85	3833	6.64
dlx-pc	39833	19.0	529	1.04
dlx-dmem	34320	18.8	1276	1.90
dlx-regfile	47822	35.5	2739	4.12
pp-bloaddata-a	8695	5.47	1193	1.80
pp-bloaddata	9016	5.56	4451	4.51
pp-dmem2	3167	2.24	2070	1.52

## SAT Heuristics and Completeness

A somewhat surprising observation is that some heuristics used by SAT solvers must be disabled or the method will be incomplete.

An example of this is the *pure literal* rule.

This rule looks for propositional variables which are either always (or never) negated in the CNF formula being checked.

These variables can instantly be replaced by *true* (or *false*).

However, if such a variable is an abstraction of a first-order atomic formula, this is no longer the case.

This is because propositional literals are independent of each other, but first-order literals may not be.

Such heuristics must be carefully disabled in the SAT solver.

## Non-Convexity Issues

As we have seen, some theories require additional case-splits before they can determine whether a set of literals is satisfiable.

For example, consider the following set of literals in the array theory  $\mathcal{T}_A$ :

$$\{read(write(a, i, v), j) = x, x \neq v, x \neq read(a, j)\}.$$

In order for the array decision procedure to determine that this set is inconsistent, it must first do a case-split on  $i = j$ .

SVC handled this situation by using a local rewrite of the following form:

$$read(write(a, i, v), j) = x \longrightarrow \mathbf{ite}(i = j, v, read(a, j)).$$

However, this approach cannot be applied in a setting where the formula being checked is translated to SAT initially.

Instead, whenever such a case split must be done, CVC adds new clauses to the SAT formula which correspond to the required case split.

Unfortunately, this approach sometimes leads to exponentially bad performance.

## CVC with SAT Compared to CVC without SAT

Example	cvc -sat		cvc +sat	
	Decisions	Time (s)	Decisions	Time (s)
bool-dlx1-c	?	> 10000	2522	1.14
bool-dlx2-aa	?	> 10000	792	0.81
bool-dlx2-cc-bug01	?	> 10000	573387	833
v-dlx-pc	8642456	5082	6137	6.10
v-dlx-dmem	2888268	2820	2184	3.48
v-dlx-regfile	29435	37.6	3833	6.64
dlx-pc	515	0.68	529	1.04
dlx-dmem	6031	4.50	1276	1.90
dlx-regfile	6386	5.27	2739	4.12
pp-bloaddata-a	93714	79.1	1193	1.80
pp-bloaddata	345569	338	4451	4.51
pp-dmem2	367877	338	2070	1.52

## Related Work

SRI's *ICS* prover has a prototype implementation combining it with SAT. However, ICS is not an explicating prover, so they do not have a good mechanism for dealing with redundant clauses.

HP (formerly Compaq, formerly DEC) Systems Research Center (SRC) (a group which includes Greg Nelson of Nelson-Oppen fame) is developing a system called *Verifun* which uses a similar approach to the one outlined here.

However, Verifun uses custom annotations to explicate inconsistencies, whereas CVC uses its proof-production engine to accomplish the same thing.

## Proofs and Explication

Having an automated theorem prover which produces proofs kills two birds with one stone.

First, proof-production improves robustness and increases confidence in the tool.

Second, proof-production provides a mechanism for explication which is critical to the success of the SAT-based approach just described.

To see this second point, suppose  $Sat_{FO}$  is presented with a conjunction of literals and determines that they are inconsistent.

If this determination is accompanied by a proof, then an analysis of the proof yields the required information about *which* literals contributed to the inconsistency.

In CVC, proof-production has two settings: a slow and thorough mode, in which full proofs are produced, and an *assumption-tracking only* mode, in which the proof-production machinery is used to track just enough information to make explication possible.

## A Framework for Producing Proofs

Proofs are represented in CVC by *sequents*.

A sequent is a pair  $\Gamma \vdash \phi$ , where  $\Gamma$  is a set of *assumptions* and  $\phi$  is a formula.

A sequent is valid if  $\mathcal{T} \cup \Gamma \models \phi$ , where  $\mathcal{T}$  is the deductive closure of the union of all theories participating in the cooperating framework of CVC.

A *proof rule*, or *inference rule* is denoted as follows:

$$\frac{P_1 \quad \cdots \quad P_n}{C}$$

where the  $P_i$ 's are *premises* and  $C$  is the conclusion of the rule (all are sequents).

A rule is *sound* if the validity of all premises implies the validity of the conclusion.

The set of premises may be empty, in which case the rule is called an *axiom*.

A *proof* or *derivation* of a sequent  $C$  is a sequence of proof rule applications that forms a finite proof tree with  $C$  as the root and axioms on the leaves.

## A Selection of Proof Rules

CVC takes a very pragmatic approach to proof rules. The goal is to have a small trusted set of proof rules that can be independently checked.

The most basic rules are the assumption axiom, proof by contradiction (also known as *negation elimination*), implication and negation introduction, modus ponens for the  $\leftrightarrow$  operator, and the cut rule:

$$\frac{}{\phi \vdash \phi} \text{assume} \quad \frac{\Gamma, \neg\phi \vdash \text{false}}{\Gamma \vdash \phi} \neg E \quad \frac{\Gamma, \alpha \vdash \phi}{\Gamma \vdash \alpha \rightarrow \phi} \rightarrow I \quad \frac{\Gamma, \alpha \vdash \text{false}}{\Gamma \vdash \neg\alpha} \neg I$$
$$\frac{\Gamma_1 \vdash \phi \quad \Gamma_2 \vdash \phi \leftrightarrow \psi}{\Gamma_1 \cup \Gamma_2 \vdash \psi} \text{MP} \quad \frac{\Gamma_1 \vdash \alpha \quad \Gamma_2, \alpha \vdash \phi}{\Gamma_1 \cup \Gamma_2 \vdash \phi} \text{cut}$$

## Representing Theorems in CVC

The main data structure in CVC for dealing with proofs is the *Theorem* data structure.

A *Theorem* consists of a sequent and may or may not contain an accompanying proof.

In the slow and thorough mode, *Theorem*'s contain a full proof which gives a derivation of the sequent in the *Theorem* using CVC's inference rules.

In *assumptions only* mode, a *Theorem* contains only the derived sequent.

## Implementing Proof Production

Instrumenting CVC to produce proofs is easier than might be expected.

The implementation framework for CVC (which is based on Nelson-Oppen) requires maintaining and communicating about facts which are true in the current context.

Instead of passing *formulas* around, we modify the framework to pass *Theorem*'s around instead.

Any time some reasoning step is performed, a *Theorem* is generated to encapsulate what was done.

# Implementing Proof Production

## Example

A *union/find* data structure is used to maintain equivalence classes of terms.

Normally a call to *find* would return the representative term for the equivalence class.

We modify this so that a call to *find* returns the *Theorem* that a term is equivalent to its equivalence class representative.

When equivalence classes are merged, the *Theorem*'s are combined using an inference rule for the transitivity of equality.

## Using Theorems to Generate Conflict Clauses for SAT

In **+sat** mode, CVC uses *Theorem*'s to generate conflict clauses as follows.

- Whenever SAT makes a decision, the corresponding literal is given as a new assumption to CVC.
- If CVC detects an inconsistency, it produces a *Theorem* whose sequent is  $\Gamma \vdash \textit{false}$  where  $\Gamma$  is guaranteed to be a subset of the assumptions handed to CVC.
- Because  $\Gamma \vdash \textit{false}$  is valid in CVC's theory  $\mathcal{T}$ , we know that  $\mathcal{T} \models \neg(\bigwedge \Gamma)$ .
- $\neg(\bigwedge \Gamma)$  corresponds to a disjunction of literals and can be abstracted to obtain a conflict clause for SAT.

## Integrating SAT Heuristics in CVC-lite

Though the SAT technique has been fairly successful for CVC, there are still problems with it:

- Because SAT solvers do not produce proofs, CVC cannot produce an overall proof in **+sat** mode.
- Non-convex theories can sometimes cause blow-up in **+sat** mode.
- Structural heuristics for choosing splitters are harder to apply because the structural information is lost in the translation to SAT.

*CVC-lite* is a new implementation of CVC which aims to address these issues as well as providing a flexible and robust platform for rapid prototyping and further development.

## Integrating SAT Heuristics in CVC-lite

CVC-lite takes the dual-use of proof-production to the next level by using proofs to implement all of the standard optimizations used by SAT solvers.

CVC-lite does not translate its input into CNF, but instead retains the original formula.

However, learned conflict clauses *are* maintained as in SAT and are processed using fast Boolean Constraint Propagation using watched literals, just as in SAT.

Thus, there are two different sets of formulas, and different decision heuristics can be used for each of them.

Some additional overhead is incurred to instrument the fast SAT algorithms with proofs.