Non-Clausal Satisfiability Modulo Theories

by

Philippe Suter

BSc., Computer Science
École Polytechnique Fédérale de Lausanne (2006)

Submitted to the School of Computer and Communication Sciences
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

September 2008

Author .....
School of Computer and Communication Sciences
August 15, 2008

Non-Clausal Satisfiability Modulo Theories

by

Philippe Suter

Submitted to the School of Computer and Communication Sciences
on August 15, 2008, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

This thesis presents NC(T), an extension of the DPLL(T) scheme [16, 29] for decision procedures for quantifier-free first-order logics. In DPLL(T), a general Boolean DPLL engine is instantiated with a *theory solver* for the theory T. The DPLL engine is responsible for computing Boolean implications and detecting Boolean conflicts, while the theory solver detects implications and conflicts in T, and the communication between the two parts is done through a standardized interface. The Boolean reasoning is done on a set of constraints represented as *clauses*, meaning that formulas have to be converted to conjunctive normal form before they can be processed. The process results in the addition of variables and a general loss of structure. NC(T) remove this constraint by extending the Boolean engine to support the detection of implications and conflicts on non–clausal constraints, using techniques working on graphical representations of formulas in negation normal form first described in [19, 21]. Conversion to negation normal form preserves the size and structure of the input formula and does not introduce new variables.

The above scheme NC(T) has been implemented as a tool called fSTP, where the theory T under consideration is the quantifier–free theory of uninterpreted function and predicate symbols with equality. We describe our implementation and give early experimental results.

Thesis Supervisor: Viktor Kuncak
Title: Professor

Thesis Supervisor: Vijay Ganesh
Title: Research scientist

Acknowledgments

I would like to thank my advisor at MIT, Vijay Ganesh, for his expertise and guidance during this project. I thank him for letting me initially take my time to choose a topic, for his constant availability and for his pleasant habit of turning every setback into a valuable learning experience. I wish him the best of luck for his work and hope to collaborate with him again in the future. I thank my advisor at EPFL, Viktor Kuncak, for giving me the opportunity to come to MIT to do this research, and look forward to working with him as I return to Switzerland. I also thank Martin Rinard for accepting me as a visiting student in his group. I thank Himanshu Jain at CMU for sharing with us his ongoing work on non-clausal reasoning and for discussing this with us at various occasions. I thank Karen Zee and Michael Carbin with whom I shared an office for providing a great working environment, and particularly the latter for continuously insisting that I should discover more of Boston and helping me doing so. I also thank the support staff at EPFL and MIT who helped me deal with their respective administrations in the best possible way; Sylviane Dal Mas and Catherine Vinckenbosch at EPFL, and Mary McDavitt and John Merriman at MIT.

I thank the many great persons I met here for making this experience more than a research internship, with highlights including squash games, the rooftop parties, Wally's, and the Sunday Morning Brookline Brunch. While certainly no list of names can pretend to completely express my gratitude and respect for these people, not attempting would be worse; Chee, Christoph, Georg, Mariam, Michael ($\times 2$), Pablo, Patrick, Sara, Sarra and Till, thanks. Finally I thank those who continuously supported me from Europe; my family, my friends, and of course Violaine.

Contents

Chapter 1

Introduction

The problem of determining the Boolean satisfiability of a formula, SAT, is central in complexity theory. This stems from its NP-complete status and from its simple and concise description, making it a suitable target for reductions from more complex problems. As a consequence, many problems can be reduced to SAT and subsequently worked on using existing, well-sustained tools, through simple interfaces.

The problem of determining the satisfiability of formulas in quantifier-free first-order theories with equality, *satisfiability modulo theories*, or SMT, extends the scope of applicability of SAT by allowing reasoning about non-Boolean values such as integers, real numbers, bit-vectors or uninterpreted function symbols, for instance.

The development of efficient tools for both these problems, called SAT solvers and SMT solvers, or *decision procedures*, has enabled a broad range of applications in formal methods, program analysis and AI. These include for instance bug-finding [6, 8], symbolic and bounded model-checking [5, 9], proofs of correctness for compiler optimizations [23], or the verification of microprocessors [36].

As applications target more challenging domains, they in turn generate more difficult instances of SAT and SMT problems. Consequently, the demand for better solvers continues to increase, driving the research into efficient decision procedures.

SAT solvers traditionally work on formulas in *conjunctive normal form*, or CNF. This standard form is easy to reason about and has been thoroughly studied in the context of DPLL SAT solvers (see Chapter 2 for details). However, this approach

requires that problems which are not naturally best expressed as a set of clauses be translated to CNF first, for instance using the well-known Tseitin encoding [35]. These transformations introduce new variables and can destroy the structure of the original problem, sometimes making it artificially hard to solve. We refer the reader to Thiffault et al. [34] and Jain et al. [20] for a detailed discussion of this problem.

Modern SMT solvers suffer from a similar problem. These solvers use a generalization of the DPLL algorithm known as DPLL(T) [16, 29] (see Chapter 2 for details). This generalization similarly requires that the Boolean structure of the input formulas be in conjunctive normal form, and consequently suffers from the same problems as in the case of SAT solvers, namely that important structural information is lost in the process.

Therefore, it is of great interest to the SAT and SMT communities, as well as the users of their tools, to come up with techniques that do not necessarily require translation of the Boolean structure of the formulas into conjunctive normal form. Furthermore, these techniques should be able to exploit the structural information to deliver greater performance.

The focus of this thesis is to propose a solution to the problem of solving SMT formulas without requiring conversion to CNF. The following section gives an overview of our solution and its implementation.

1.1 Proposed solution

The first contribution of this thesis is a proposal to extend the DPLL(T) scheme by adding to it an extra component allowing reasoning on non-clausal constraints, thus forming a new scheme that we call NC(T). Just like in DPLL(T), a NC(T)-based decision procedure consists of a theory solver for the theory T and a general, theory-unaware, Boolean solver. We preserve the standard interface for theory solvers described in [16]. Unlike in DPLL(T), our proposal for the Boolean solver is a combination of a standard DPLL solver, working on constraints expressed as clauses, and a non-clausal solver, based on [19, 21], capable of reasoning about constraints expressed

as formulas in negation normal form (NNF). Particularly important in our design is the communication between these two components: new implications and conflicts discovered by the non-clausal reasoner are encoded as clauses and communicated to the clausal part. This results in an *incremental translation* into conjunctive normal form: as the number of clauses augments, so does the proportion of conflicts and implications found during clausal reasoning against those found on the non-clausal constraints. This approach enables us to leverage the efficient existing techniques for DPLL/DPLL(T) while at the same time avoiding the initial translation to CNF.

Our second contribution is fSTP, an implementation of the described scheme along with a theory solver for the quantifier-free theory of uninterpreted function and predicate symbols with equality ($T^{=}_{\mathsf{UF}}$). The implementation of our theory solver is based on [16, 27, 28].

1.2 Organization of this thesis

The rest of this thesis is organized as follows: Chapter 2 gives a quick overview of the design of modern DPLL-based SAT solvers and DPLL(T)-based SMT decision procedures. Chapter 3 examines techniques first introduced in [19, 21] for reasoning on non-clausal constraints and shows how they can be used to extend the general DPLL(T) scheme to form the new NC(T) scheme. Chapter 4 describes fSTP, our implementation of NC(T) for the theory $T^{=}_{\mathsf{UF}}$. Chapter 5 presents some experimental results we obtained by comparing our tool to an existing, DPLL(T)-based one, and Chapter 7 concludes by presenting some ideas for future work.

Chapter 2

Reasoning on clausal constraints

This chapter presents the foundations of the DPLL algorithm for SAT, and its adaptation to the problem of satisfiability modulo theories, DPLL(T). Both these applications have a record of success against competing approaches. DPLL, along with some of the popular techniques for clausal learning, fast Boolean constraint propagation, restarts, etc., is the algorithm of choice in virtually every competitive SAT solver [15, 17, 25]. The DPLL(T) approach combines the power of clausal solvers and a high versatility which makes it a suitable choices for many theories. SMT decision procedures based on DPLL(T) regularly perform extremely well in many categories at the annual SMT competition [3, 4, 12, 14, 26].

2.1 Conversion to CNF

As we already mentioned in the introduction, the DPLL algorithm requires that input formulas be in conjunctive normal form. Any Boolean formula can be converted to that form using for instance the distributivity laws on $\wedge$ and $\vee$, however this is not practical, as the size of the formula can then grow dramatically. A technique commonly used instead is the *Tseitin encoding* [35].

The idea behind the Tseitin encoding is to add new propositional variables to represent subformulas, introduce additional clausal constraints to link the value of these variables to the subformula they represent, and then flatten out the main formula by

replacing each subformula by its representative.

Conjunctions $p_1 \wedge \ldots \wedge p_m$ can be replaced by a single variable T_n if we add the clausal constraints $(T_n \vee \neg p_1 \vee \ldots \vee \neg p_m) \wedge (\neg T_n \vee p_1) \wedge \ldots \wedge (\neg T_n \vee p_m)$. These new constraints encode the relation $T_n \iff (p_1 \wedge \ldots \wedge p_m)$.

Disjunctions are treated dually: $p_1 \vee \ldots \vee p_m$ gets represented by T_n by adding the constraints $(\neg T_n \wedge p_1 \wedge \ldots \wedge p_m) \vee (T_n \wedge \neg p_1) \vee \ldots \vee (T_n \wedge \neg p_m)$, encoding $T_n \iff (p_1 \vee \ldots \vee p_m)$.

One can then traverse recursively the structure of the formula, flattening the inner-most subformulas first and proceeding until the top level is reached. The size of the encoded formula is linear in the size of the original one[1].

2.2 DPLL

What falls under the denomination "the DPLL algorithm" is usually more than the simple Davis-Logemann-Loveland algorithm [10], itself a variation on the David-Putnam algorithm [11]. It is assumed that all implementations will include features such as *clausal learning* and *non-chronological backtracking*, or efficient heuristics for *literal selection*, *restarts* or *clause deletion*. We briefly survey the elements among these whose understanding is required for the reading of the rest of this thesis.

A modern DPLL-based decision procedure follows the structure of Algorithm 1 [15, 25]. The idea is to attempt to determine the satisfiability of the input formula by incrementally building a satisfying assignment. Whenever a new literal is added to the assignment, or asserted, the *decision level* of the procedure is incremented, and the new assignment is checked against the formula. If some variables in the formula have to assume a certain polarity to maintain the overall satisfiability as a result of the assertion of new literals, the combination of these variables and their polarity forms new *implied literals* at the current decision level. If on the other hand the assignment makes the formula unsatisfiable, one reached a *conflict* situation. When a conflict occurs, the cause of that conflict is determined and a certain number of the

[1] As long as we restrict ourselves to the Boolean operators $\wedge$ and $\vee$.

Algorithm 1 DPLL-like decision procedure

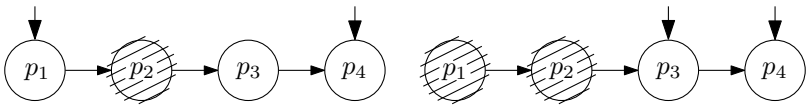
```
1: while ⊤ do
2:     if AssertNextLiteral() = ⊤ then
3:         Propagate()
4:         if FoundConflict() = ⊤ then
5:             if AnalyzeConflict() = ⊥ then    ▷ Conflict could not be resolved
6:                 return unsat
7:             else
8:                 Backtrack()
9:             end if
10:        end if
11:    else                                     ▷ All variables were assigned
12:        return sat
13:    end if
14: end while
```

latest asserted literals are removed from the assignment: this operation constitutes *backtracking* to an earlier decision level. When the analysis of a conflict determines that the search needs to be backtracked until prior to the first assertion, the conclusion is that the formula under consideration is unsatisfiable. On the other hand, when all variables have been assigned a polarity and no conflict has been found, then the formula is satisfiable and the built assignment is a satisfying assignment for it.

2.2.1 Implications and conflicts

Being able to determine new implications from a partial assignment, also known as *Boolean constraint propagation* (BCP), and determining which assignments are conflicting are two essential parts of any DPLL-based decision procedure. A technique introduced in [25] known as the *two-watched-literal* scheme is what is commonly used to perform both these tasks. The principle works as follows:

- In each clause, two literals are "watched", and we try to maintain as an invariant that none of them is falsified by the current assignment.
- Whenever one of these two literals becomes falsified, we have three possible cases:



(a) Before the assignment of $\bar{p_1}$ (b) After

Figure 2-1: Example of two-watched-literal scheme application

1. There exists another, non-watched literal in the clause which is not falsified. In that case, we stop watching the falsified literal and start watching that other one. There are no implications and we are not in a conflicting situation.
2. The only non-falsified literal is the one being watched by the second watch. In that case, that literal is *implied* by the current assignment. Indeed, it has to be satisfied, or the clause would then be entirely falsified.
3. All literals in the clause are already falsified. In that case, we have reached a conflict.

The two-watched-literal scheme presents the double advantage that it is fast, as each time a new literal is asserted, only the small set of clauses where that literal is one of the two being watched has to be examined, and no operation needs to be done when the search backtracks to a previous decision level. Indeed, all clauses in which watches are set on non-falsified literals will conserve that property, as backtracking only cancels assignments.

Figure 2-1 shows an illustration of the two-watched-literal scheme: before $\neg p_1$ gets asserted, both watches are on non-falsified literals. When $\neg p_1$ becomes true, p_1 becomes false and the watch has to be moved to another non-falsified literal. After this point, if p_3 or p_4 becomes false, then the other one is implied. If on the other hand they both become false at the same decision level, for example because $\neg p_3$ triggers $\neg p_4$ in a different clause, then the assignment is conflicting.

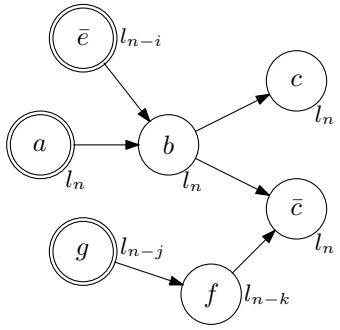


Figure 2-2: Example of an implication graph. Double circles denote decision literals. Decision levels are indicated next to the literals.

2.2.2 Conflict resolution and clause learning

Whenever a literal is implied, the clause which triggered the implication is associated to it. This allows us to retrieve the *reason* of an assignment; each literal is assigned either because it was the one asserted at its decision level, or because a set of already assigned literals triggered its implication in a clause. One can visualize this information as a directed graph, which we call an *implication graph*, where decision literals are root nodes and BCP-implied literals have incoming edges from the other literals in their clause. Consider for instance the formula:

$$(\neg a \vee b \vee e) \wedge (\neg b \vee c) \wedge (\neg f \vee \neg b \vee \neg c) \wedge (\neg g \vee f) \tag{2.1}$$

under the partial assignment $\{a, \neg e, g\}$. The assignment and its implications are graphically depicted in Figure 2-2. b is implied by a and $\neg e$ and because of the first clause, f is implied by g and because of the first clause, and both c and $\neg c$ are implied, because of the second and third clauses respectively. This is a conflicting situation.

The simple strategy for conflict resolutions of the original DLL algorithm is as follows: whenever a conflict is reached, one searches for the latest asserted literal which is not marked as having been tried in both polarities. One backtracks to the decision level of that literal, flips its polarity in the assignment, marks it as flipped and then continues the search from there. Modern approaches however *use* conflicts rather than just *working around* them: when a conflict is reached, a new clause

which records the cause for it is added to the clause set. This avoids falling in similar conflicting situations at later stages.

2.2.3 First unique implicant

A conflict arises when both the true and false polarities get assigned to a variable. This can be observed in the example depicted in Figure 2-2. One can learn the *reason* for such a conflict by following the edges in the implication graph backwards from the two nodes containing the variable in opposed polarities, as the set of nodes which have outgoing edges to another one represent the set of literals responsible for the implication. We can use such a set to produce a *learnt clause* which can then be added to the clause set the ensure that the same conflict will not be triggered again. In our example, we could add the clause $\neg b \vee \neg f$ to the set. This will have the desired effect that the next time b is asserted or implied, $\neg f$ will be found as a consequence of BCP and the same conflict will not arise again.

$\neg b \vee \neg f$ is not the only clause that we could learn. We could for instance replace b by the set of literals that implied b, creating instead the clause $e \vee \neg a \vee \neg f$. It is not difficult to see that we could keep going, replacing literals by their implicants until all literals in the clause are asserted literals, which by definition have no implicants. The question of which type of clause to learn has been well studied [37], and the strategy generally chosen works as follows: when a conflict is reached at decision level l, we first compute the set of the direct implicants, then we incrementally replace all literals from the decision level l in the set by their own implicants until there is only one of them left[2]. The goal of this strategy, called *first unique implicant*, is to learn a clause where there is only one literal from the decision level at which the conflict occurs. We then search for the second-highest decision level among the literals in that clause, and backtrack the search to that level. As only the unique implicant will then become unassigned and all other literals in the clause will be falsified, BCP will trigger its new assignment.

[2]Note that there is always necessarily at least one, namely the literal asserted at that decision level.

We conclude this section on DPLL by quickly presenting techniques commonly found in solvers. The main concern of this thesis is not DPLL, though, and we do not elaborate on these.

2.2.4 Dynamic literal selection

An important component of an efficient SAT solver is a good heuristic for the selection of the next literal to assert. Most solvers use variations of the VSIDS heuristic [25], where scores are assigned to all variables in both polarities based on the number of clauses they appear in. When new clauses are added, the score for the literals in them augments, improving the chance that they will be picked soon. Periodically, all scores are divided by a constant. The rationale behind this technique is that the heuristic will tend to satisfy learnt clauses, with a particular focus on recent ones. See [32, 37] for details.

2.2.5 Restarts

Another feature commonly found in SAT solvers are automated *restarts*: periodically, for instance after a fixed number of decisions or learnt clauses, the search restarts from scratch, although the set of learnt clauses is conserved. This is done in order to prevent the search from getting stuck in subspaces where no substantial progress can be made. More details can be found in [18].

2.2.6 Clause management

Finally, most solvers put bounds on the size of the clause set they maintain, to avoid an explosion in memory usage. Techniques vary among implementations; some solvers delete clauses only during restarts, some merge clauses or attempt to detect situations where some clauses are subsumed by other one, etc. More information can be found in [15, 25].

2.3 DPLL(T)

We now move on to the description of the DPLL(T) scheme [16, 29]. We start by informally describing the theory to which we will bring our focus for the rest of this thesis, $T^{=}_{\mathsf{UF}}$. We will then use it as an example to describe the main features of DPLL(T).

2.3.1 $T^{=}_{\mathsf{UF}}$

The *quantifier-free theory of uninterpreted function symbols and predicates with equality*, which we denote by $T^{=}_{\mathsf{UF}}$, is also sometimes called the *empty theory*. Valid terms in that theory are *uninterpreted constants* (a, b, ...) and applications of *uninterpreted functions* and *uninterpreted predicates* of arbitrary arity ($f(a)$, $p(a, b)$, $f(f(a))$, ...). Valid atoms are equalities between uninterpreted terms and function applications, and predicates or applications of uninterpreted predicate symbols. Thus, some examples of literals in $T^{=}_{\mathsf{UF}}$ are for instance $f(a, b) = c$, $\neg p(c)$ or $a \neq b$.

Equality is reflexive, transitive and symmetric, and the additional *congruence axiom* applies. For a function symbol f of arity n, we have that:

$$a_1 = b_1 \wedge \ldots \wedge a_n = b_n \implies f(a_1, \ldots, a_n) = f(b_1, \ldots, b_n) \tag{2.2}$$

Similarly for a predicate symbol of arity n:

$$a_1 = b_1 \wedge \ldots \wedge a_n = b_n \implies (p(a_1, \ldots, a_n) \iff p(b_1, \ldots, b_n)) \tag{2.3}$$

The satisfiability problem for a formula ϕ in $T^{=}_{\mathsf{UF}}$ consists in determining whether there exists a *congruence relation* on all uninterpreted constants in ϕ such that ϕ is true whenever '=' is interpreted as this congruence relation. A congruence relation is a set of congruence classes, where each uninterpreted constant appears in exactly one class, and all terms in a class are defined to be equal. For instance, the formula:

$$f(a) = f(b) \wedge a \neq b \wedge b = c \tag{2.4}$$

is satisfied by the congruence relation $\{\{a\}, \{b, c\}\}$, while the formula:

$$f(a, b) = a \wedge f(f(a, b), b) = b \wedge a \neq b \tag{2.5}$$

admits no satisfying congruence relation.

2.3.2 Theory solver interface

DPLL(T)-based decision procedures are built around a DPLL Boolean solver and a decision procedure for conjunctions of literals in the theory T. Decision procedures for conjunctions of literals of $T^{=}_{\mathsf{UF}}$ are well-studied [13, 28], and we do not describe them here. During solving, the Boolean solver asserts new literals, as in standard DPLL, and communicates them to the theory solver. The theory solver maintains a stack of asserted literals. To be properly usable in the context of DPLL(T), it must also implement some operations which are described as a standardized interface [16]. We present this interface along with examples for a solver for $T^{=}_{\mathsf{UF}}$:

- The Boolean solver can tell the theory solver to *assert* a new literal. The theory solver adds that literal to its stack, and either responds by saying that the new set of literals is inconsistent, or by sending back a list of literals which are *implied* by the latest addition. For instance, if the stack already contains the literals $a = b$ and $c = f(c)$, then the assertion of $f(a) \neq f(b)$ will make the set inconsistent. On the other hand, the assertion of $b = c$ would produce the newly implied literals $a = f(c)$ and $b = f(b)$, for instance. Note that the set of literals is not required to be complete.

- The Boolean solver can query the theory solver with a literal to ask it whether it is implied by the current set of asserted literals. In our previous example, a query for $f(a) = f(b)$ would return "true" while a query for $b = c$ would return "false". Note that a "false" response does not mean the negation of the literal is implied.

- The Boolean solver can order the theory solver to backtrack to a previous decision level, thus popping some elements off the stack.

- The Boolean solver can ask for an *explanation* for a true literal l, consisting in a subset of the asserted stack which logically implies l. For instance, if the stack already contains the literals $a = b$, $f(a) = a$ and $b = c$, and if a query for an explanation for $g(a) = g(c)$ is made, then the subset $\{a = b, b = c\}$ would be returned, not containing the irrelevant literal $f(a) = a$.

- Finally, when the set of asserted literals is inconsistent, the Boolean solver can ask for an *unsat core*, a subset of literals from the stack whose conjunction is unsatisfiable.

In the case of $T^{=}_{\mathsf{UF}}$, [28] and [26] describe how a solver implementing this interface can be built.

We can now informally describe the workings of a DPLL(T) decision procedures. Just as in DPLL, the goal is to incrementally build a satisfying assignment by successively asserting literals. Each time a new literal is asserted, it is passed to the theory solver.

New literals can also be discovered through Boolean implications, as in DPLL, or through theory implications, returned by the theory solver as explained in the interface description. The method for retrieving the implicants of a literal therefore depends on the type of implication: for Boolean implications, this is done by examining the implicant clause, as in DPLL, and for theory implications by querying the theory solver for explanations. As a consequence, the implication graphs in such decision procedures contain two types of edges. Figure 2-3 shows such a graph, where dashed edges indicate theory implications.

Conflict analysis is done as in DPLL, with the computation of a first unique implicant and the subsequent backtracking. The only different cases are *theory conflicts*, where the reason for the conflict is a theory-inconsistent set of asserted literals rather than a Boolean contradiction. When such a situation arises, the theory solver is asked

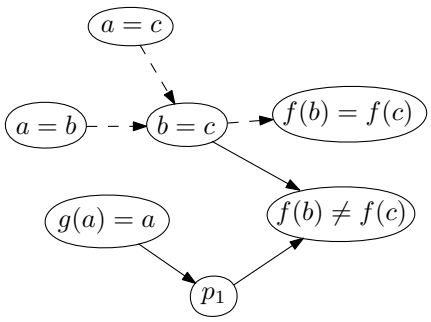


Figure 2-3: Example of an implication graph including theory implications

to return an unsat core, which is then used to form a clause. We then use the first unique implicant technique on that clause.

Most other important features from DPLL such as the VSIDS heuristic or the use of restarts are used in DPLL(T) as well.

Our description of DPLL(T)-based decision procedures is certainly over-simplistic. We believe however that it is sufficient to understand our proposal for an extension of this scheme, that we present in the next chapter. We refer the reader to [29] for a complete account of the theoretical foundations of DPLL(T).

Chapter 3

Reasoning on non-clausal constraints

In the previous chapter, we have given an overview of the workings of SAT and SMT solvers on constraints expressed in *conjunctive normal form*. We have seen how efficient algorithms can be implemented to perform two fundamental tasks on these formulas; *conflict detection* and *Boolean constraint propagation*. We now bring our focus to constraints expressed as arbitrary formulas. We show how the conflict detection and Boolean constraint propagation operations can be effectively performed on these constraints, after a reduction to a different, simpler normal form.

3.1 Graph-based representation of non-clausal formulas

3.1.1 Negation normal form

In this chapter we assume that Boolean formulas contain only $\wedge, \vee, \neg$ as propositional operators.

A Boolean formula is in *negation normal form* (NNF) if the negation operator $\neg$ appears only directly in front of variables. In other words, a formula is in NNF if it consists only of literals connected together with the Boolean operators $\vee$ and $\wedge$.

Any formula can be mechanically transformed into a logically equivalent formula in NNF by applying De Morgan's laws and removing double negations.

For example, the formula:

$$((p \vee q) \wedge \neg r \wedge \neg(p \wedge q)) \vee \neg(p \vee (\neg s \wedge (r \vee s)) \vee \neg q) \tag{3.1}$$

is logically equivalent to its negation normal form:

$$(p \vee q) \wedge \neg r \wedge (\neg p \vee \neg q)) \vee (\neg p \wedge (s \vee (\neg r \wedge \neg s)) \wedge q) \tag{3.2}$$

The process naturally extends to formulas in first-order theories. For instance, the formula in $T^{=}_{\mathsf{UF}}$:

$$f(a, b) = a \wedge \neg(a = b \vee f(f(a, b), b) \neq b) \tag{3.3}$$

has the following equivalent in NNF:

$$f(a, b) = a \wedge a \neq b \wedge f(f(a, b), b) = b) \tag{3.4}$$

Two valuable properties of negation normal form are that the normalization process does not introduce new variables and does not increase the size of the original formula, if we make the reasonable assumption that positive and negative literals can be represented using the same amount of space.

3.1.2 Two-dimensional diagrams

In [1], Andrews introduces a two-dimensional representation for formulas in NNF. Following his convention, we write disjunctions in the usual, horizontal, fashion while we represent conjunctions by displaying their conjuncts in vertical stacks surrounded by brackets. As an example, Figure 3-1 shows (3.2) in its two-dimensional representation. Andrews goes on by defining *vertical paths* in such representations: informally, one obtains a vertical path by traversing the diagram from top to bottom choosing

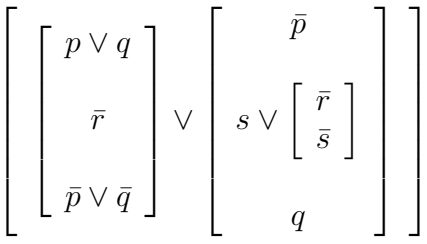


Figure 3-1: Two-dimensional representation of (3.2). We use $\bar{p}$ to denote $\neg p$.

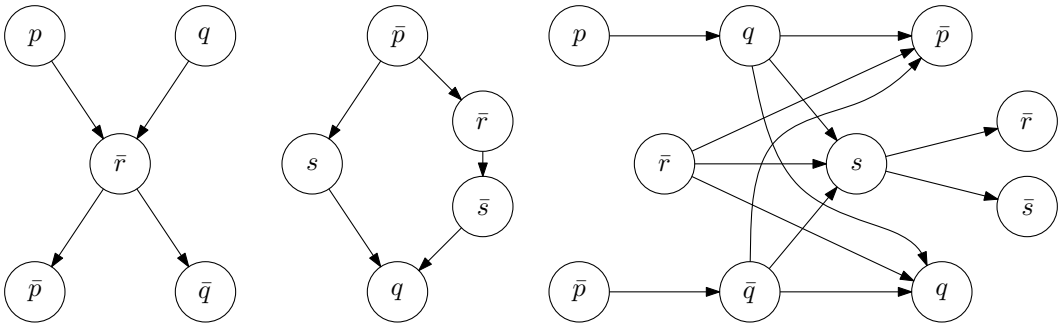
for each encountered disjunction to traverse one of the disjuncts. Possible vertical paths in (3.2) are for instance $p \to \neg r \to \neg p$ or $\neg p \to \neg r \to \neg s \to \neg q$. *Horizontal paths* are defined in a similar, dual, way. Examples of horizontal paths in Figure 3-1 are $p \to q \to s \to \neg s$ or $\neg r \to \neg p$.

3.1.3 Vertical and horizontal graphs

A perhaps more explicit way to display these paths is proposed in [20]; we construct directed acyclic graphs where nodes contain literals and edges represent connections in the two-dimensional formula representation. For a formula ϕ, we can build a *vertical graph* (vGraph) $G_V(\phi)$ and a *horizontal graph* (hGraph) G_H. Such graphs are defined as tuples $G = (V, E, R, L, c)$ where:

- V is the set of vertices. $|V|$ is equal to the number of literals in ϕ, multiple occurences of the same literal being counted multiple times.
- $E \subseteq V \times V$ is the set of edges
- $R \subseteq V$ is the set of nodes with in-degree 0 (roots)
- $L \subseteq V$ is the set of nodes with out-degree 0 (leaves)
- $c : E \to l$ is a function which maps each node in the graph to a literal

We will consider pairs of vertical and horizontal graphs $G_V(\phi) = (V_V, E_V, R_V, L_V, c_V)$ and $G_H(\phi) = (V_H, E_H, R_H, L_H, c_H)$ such that $V_V = V_H$ and $c_V = c_H$. Figure 3-2 shows the vGraph and hGraph for (3.2).



(a) vGraph (b) hGraph

Figure 3-2: Graph representations of (3.2)

Algorithm 2, adapted from [20], presents a mechanical procedure to build a vGraph for a given formula. Informally, the process consists of recursively building graphs for the subformulas, starting with the inner-most single literals and acting appropriately on the Boolean operators $\vee$ and $\wedge$:[1]

- For disjunctions, the *union* of the graphs representing the two sub-formulas is taken. Visually, this corresponds to "placing the two graphs next to each other", without adding any new edge between their nodes.

- For conjunctions, the *concatanation* of the graphs is computed. This corresponds to "placing the graphs on top of each other", and connecting each leaf node of the top graph to every root node of the bottom one.

The process is non-deterministic in the sense that the operands of $\vee$ and $\wedge$ can be picked in any order, resulting in different graphs. Figure 3-3 for instance shows an alternative vGraph for (3.2). The set of vertical paths, however, is unique for a given formula, if we ignore the ordering of literals in the path.

A similar procedure can be dually defined for the construction of hGraphs, where subgraphs are concatenated for disjunctions and united for conjunctions.

Andrews showed that for any Boolean formula ϕ, writing each vertical paths in $G_V(\phi)$ as a conjunct and taking the disjunction of all of them yields a new formula in disjunctive normal form ϕ_{DNF}, with the property that ϕ and ϕ_{DNF} are equivalent. For

[1]Note that we consider the operators as *binary* here.

Algorithm 2 Vertical graph construction

1: **function** BUILDVGRAPH(ϕ)
Require: ϕ is in NNF
2: **if** ϕ is a single literal l **then**
3: **return** $(\{v_i\}, \{v_i\}, \{v_i\}, \emptyset, \{v_i \mapsto l\})$ ▷ where i is a fresh index
4: **else if** ϕ is $\phi_1 \vee \phi_2$ **then**
5: $(V_1, E_1, R_1, L_1, c_1) \leftarrow$ BUILDVGRAPH(ϕ_1)
6: $(V_2, E_2, R_2, L_2, c_2) \leftarrow$ BUILDVGRAPH(ϕ_2)
7: **return** $(V_1 \cup V_2, E_1 \cup E_2, R_1 \cup R_2, L_1 \cup L_2, c_1 \cup c_2)$
8: **else if** ϕ is $\phi_1 \wedge \phi_2$ **then**
9: $(V_1, E_1, R_1, L_1, c_1) \leftarrow$ BUILDVGRAPH(ϕ_1)
10: $(V_2, E_2, R_2, L_2, c_2) \leftarrow$ BUILDVGRAPH(ϕ_2)
11: **return** $(V_1 \cup V_2, R_1, L_2, E_1 \cup E_2 \cup (L_1 \times R_2), c_1 \cup c_2)$
12: **end if**
13: **end function**

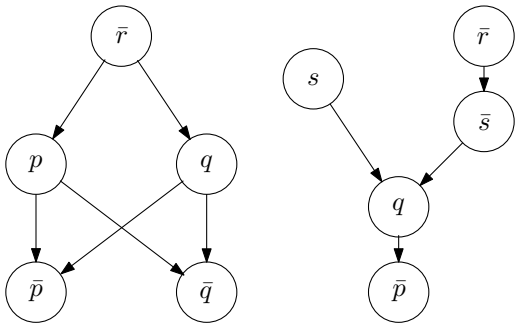


Figure 3-3: Alternative vGraph for (3.2)

instance, by enumerating the vertical paths in Figure 3-2(a), we can rewrite (3.2) as:

$$\begin{aligned}(p \wedge \neg r \wedge \neg p) \vee (p \wedge \neg r \wedge \neg q) \vee (q \wedge \neg r \wedge \neg p) \\ \vee (q \wedge \neg r \wedge \neg q) \vee (\neg p \wedge s \wedge q) \vee (\neg p \wedge \neg r \wedge \neg s \wedge q)\end{aligned} \tag{3.5}$$

Or we can rewrite it in CNF by enumerating the paths in Figure 3-2(b):

$$\begin{aligned}(p \vee q \vee \neg p) \wedge (p \vee q \vee s \vee \neg r) \wedge (p \vee q \vee s \vee \neg s) \wedge (p \vee q \vee q) \\ \wedge (\neg r \vee \neg p) \wedge (\neg r \vee s \vee \neg r) \wedge (\neg r \vee s \vee \neg s) \wedge (\neg r \vee q) \wedge (\neg p \vee \neg q \vee \neg p) \\ \wedge (\neg p \vee \neg q \vee s \vee \neg r) \wedge (\neg p \vee \neg q \vee s \vee \neg s) \wedge (\neg p \vee \neg q \vee q)\end{aligned} \tag{3.6}$$

3.1.4 A simple algorithm for satisfiability

The previous fact leads to a very simple algorithm for deciding the satisfiability of a Boolean formula ϕ, presented in [1]:

- Convert ϕ to NNF
- Build its vGraph
- Enumerate all vertical paths
- If there is one such that no literal appears positively *and* negatively in it, then the formula is satisfiable, otherwise it is unsatisfiable.

The correctness of the algorithm follows directly from the observation that the set of vertical paths corresponds to a DNF encoding of ϕ; a path where no literal appears in both polarities is a conjunct which can be used to build a (partial) satisfying assignment for ϕ by assigning to each literal in the path the truth value corresponding to its polarity. This trivially satisfies the conjunction corresponding to the path, satisfying in turn the whole DNF form. On the other hand, if each path contains at least one literal and its negation, then any assignment will falsify all conjunctions.

This decision procedure can be adapted to the problem of satisfiability modulo theories: rather than searching for a vertical path with no self-contracting literal,

one simply searches for one which is *theory-consistent*, that is where no subset of the literals from the path implies the negation of another one.

While correct, this algorithm is not practical, as the number of vertical paths can be exponential in the size of the formula. We will now describe an alternative approach, working on the same graphical representation of formulas, and which, along with DPLL(T), forms the basis of the NC(T) framework.

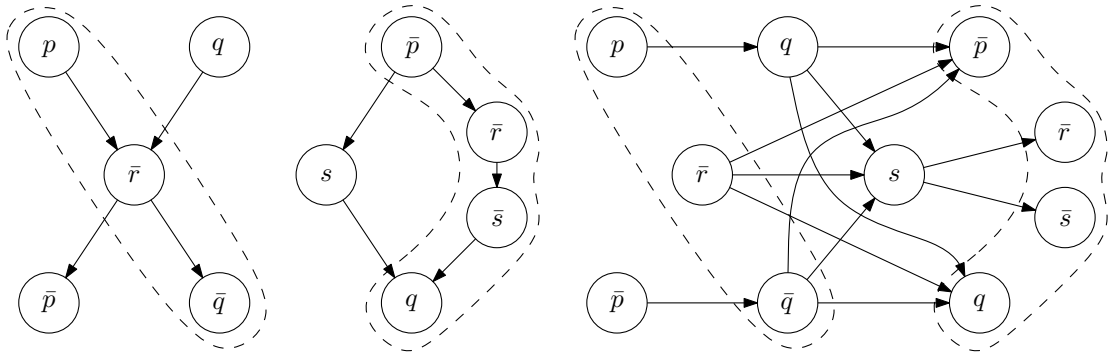
3.2 Application to DPLL-based solvers

In Chapter 2, we have given an overview of the main features of DPLL-based decision procedures; a satisfying assignment is incrementally built by searching through the space of all possible assignments, and that space is pruned by performing *Boolean constraint propagation* and *conflict detection* on a set of constraints. In conventional DPLL (and DPLL(T)), these constraints are necessarily encoded as clauses. This section presents techniques to do BCP and conflict detection on constraints represented by NNF formulas.

These two operations are made very efficient in DPLL thanks to the *two-watched-literal scheme* [25], which essentially allows to discover implications of the currently considered partial assignment while only looking at a small subset of the literals in the clauses. Jain et al. introduce in [19, 21] a similar technique for non-clausal constraints, which we present here. The theory is his contribution, while the displayed algorithms and the integration into the broader NC(T) framework is our work.

3.2.1 Terminology

We define a *cut* as a set of nodes in a vGraph such that every horizontal path in a hGraph for the same formula has a least one node in that set. It is not difficult to show that any vertical path constitutes such a set [21]. Figure 3-4 shows two possible cuts in a hGraph and the corresponding vGraph. We say a literal l is in a cut if there exists a node v in that cut such that $c(v) = l$. Given a partial assignment, a cut can be *acceptable* if none of the literals it contains are falsified by the assignment, or



(a) vGraph (b) hGraph

Figure 3-4: Two cuts, shown on the vGraph and the hGraph of the same formula

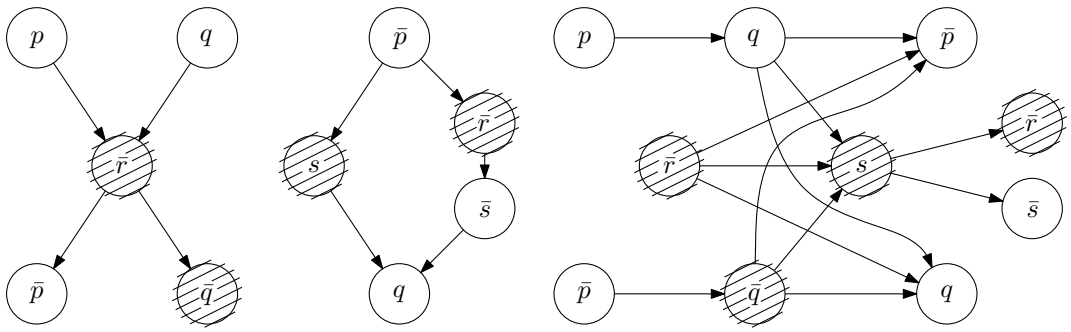
not acceptable if at least one of the literals is present with the polarity opposed to its assigned truth value. Whether or not a cut contains a literal and its negation is not a criterion for acceptability. Furthermore, cuts can be *overlapping* if they share a subset of common nodes. Having common literals is not a sufficient condition for overlapping. In Figure 3-4 for instance, both cuts contain a node representing the literal $\neg r$ but they are distinct. These cuts are not overlapping.

Cuts can play a role very similar to watches literals on clauses. As an invariant during the search, we will always try to maintain two acceptable, non-overlapping cuts. When this is not possible, we have either reached a conflict or found new Boolean implications. We will now examine both scenarios.

3.2.2 Conflict detection

A partial assignment is conflicting when no acceptable cut can be found. Intuitively, this corresponds to a situation where the current assignment falsifies all conjunctions in the DNF equivalent of the formula. Figure 3-5 gives an example of such situation, where the partial assignment $\{r, \neg s, q\}$ leads to a conflict.

When conflicts arise, an examination of the hGraph can reveal their cause. Remember that the horizontal graph compactly encodes the set of clauses in the CNF representation of the formula. For an assignment to be conflicting, it has to completely falsify at least one of these clauses, and it can be found by searching for an



(a) vGraph (b) hGraph

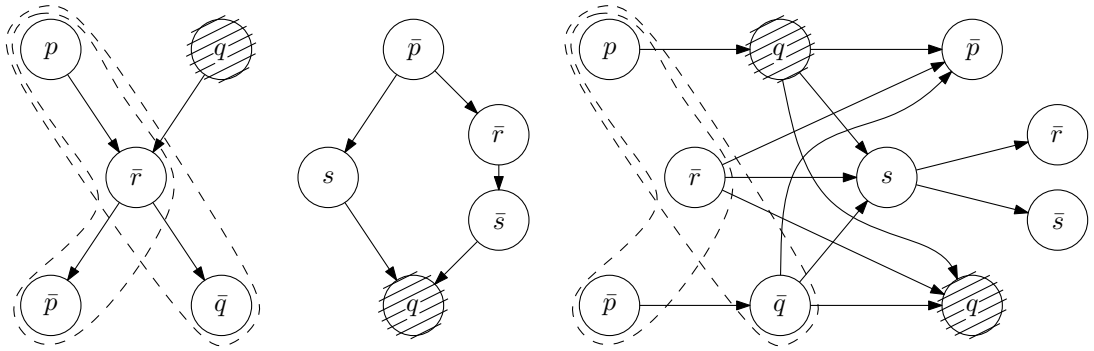
Figure 3-5: Conflicting assignment

horizontal path that is completely falsified. In Figure 3-5(b) for instance, the path $\neg r \rightarrow s \rightarrow \neg r$ meets that criterion. We can therefore conclude that the clause $\neg r \vee s$ plays a direct role in the conflict. This is not a proper newly *learnt clause*, as it was already present in the graphical encoding, but its extraction is useful for conflict analysis: we can extend it to include a unique implicant as we saw in Chapter 2 and then backtrack to the appropriate level, just as any other conflict clause in standard DPLL.

3.2.3 Boolean constraint propagation

When the only acceptable cuts we can find are overlapping, we have reached a situation where some literals in the formula are logically implied by the current partial assignment. The reason for this lies once again in the clausal encoding of the hGraph. It can only be the case that the remaining acceptable cuts have to overlap when there is at least one horizontal path such that all literals in it but one are falsified by the current assignment. Indeed, if all horizontal paths contained at least two literals which were not falsified, then one could build two cuts by following a vertical path through precisely these literals.

Figure 3-6 shows an example where the current partial assignment $\{\neg q\}$ has two new consequences p and $\neg r$. From both overlapping nodes containing the implied literals, we can build a horizontal path containing only falsified literals but one, and from these paths obtain an implicant clause. p is on the path $p \rightarrow q \rightarrow q$, yielding



(a) vGraph (b) hGraph

Figure 3-6: Example of Boolean implications

the clause $q \vee p$, and $\neg r$ is on the path $\neg r \rightarrow q$, yielding the clause $\neg r \rightarrow q$.

3.2.4 Relation to the two-watched-literal scheme

As Jain et al. properly demonstrate [19, 21], the *two-watched-cut scheme*, as we could call it, is in fact a generalization of the two-watched-literal idea; indeed, the graphical representation of a formula in CNF is as follows: the hGraph is a set of disconnect paths, where no node as an in-degree or an out-degree greater than one, and of these paths represents a clause. The vGraph connects each clause to exactly one another one, meaning that all node in a clause have exactly the same incoming and outgoing edges. The two-watched-literal scheme requires that two nodes be watched in each clause[2]. The set of all watches thus necessarily forms two vertical paths in the vGraph. An overlap of these cuts corresponds to a situation where only one watch can be put on a non-falsified literal, meaning the literal in question is implied. The absence of any acceptable cut on the other hand arises from a situation where a clause is entirely falsified.

[2]Two *literals*, in fact, but it also requires that the clauses be pre-processed so that they do not contain multiple instances of the same atom.

3.2.5 Algorithms

We have described the use of watched cuts to perform BCP and conflict detection. It remains to be explained how we can find an acceptable first cut, and how we can then proceed to find a second *least-overlapping* cut. We are, at the time of writing, unaware of the implementation details in Jain's work, however private conversations reveal that they must be similar to what we present.

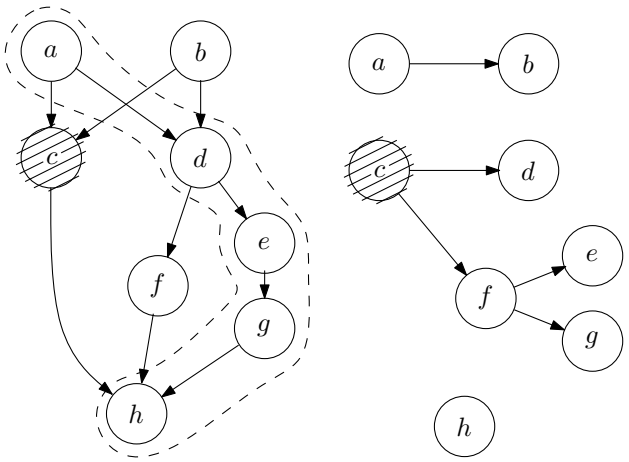
Finding a first acceptable cut is rather straightforward: we essentially perform depth-first search on the vertical graph starting with the root nodes and proceeding until a leaf node is encountered. Whenever we find a node containing a falsified literal, we backtrack. Since we are working on a directed acyclic graph and not a tree, it may be that the search traverses the same nodes several times. To avoid that, we mark nodes containing falsified literals as "dead", and we also do so recursively for nodes with only dead successors. Algorithm 3 details the procedure for the vGraph of a formula ϕ and a partial assignment A.

The search for the minimally overlapping second cut is a little more subtle. We essentially proceed by a depth-first search again, only this time we not only mark and avoid the dead nodes, but also the nodes leading only to nodes which are either dead or part of the first cut. Since we only build this second cut if we could successfully find the first one, we know it will always be possible to complete it: in the worst case, all nodes will be overlapping and we will simply build the first cut again. Another important consideration is that we can reuse the markings made in the first search to avoid spending time searching through dead nodes. With this in mind, we can informally describe the procedure: starting with the root nodes, we try to recursively find a successor node which is neither dead, nor leading to nodes in the first cut. If we cannot find such a node, then there is either exactly one non-dead node, and it is part of the first cut, or there are also non-dead nodes which are not part of the first cut but lead only to nodes which are. If such nodes exist, we chose any of them and continue from there, otherwise, we pick the node in the existing cut.

We take the example depicted in Figure 3-7: a first acceptable cut was found, and

Algorithm 3 Search for an acceptable cut

```
1: function CutSearch(G_V(φ), A)
2:     (V, E, R, L, c) ← G_V(φ)
3:     ∀v_i ∈ V, marked(v_i) ← ⊥
4:     (ok, p) ← CompletePath(R, ∅)
5:     if ok = ⊤ then
6:         return p
7:     else
8:         return ⊥
9:     end if
10: end function
11: function CompletePath(V, p)
12:     foreach v ∈ V do
13:         if marked(v) = ⊤ ∨ c(v) is false in A then
14:             continue
15:         end if
16:         S ← {w|(v, w) ∈ E}                    ▷ Successors of v in the graph
17:         if S = ∅ then
18:             return (⊤, p)
19:         end if
20:         (ok, p_2) ← CompletePath(S, p → v)
21:         if ok = ⊤ then
22:             return (⊤, p_2)
23:         else
24:             marked(v) ← ⊤                     ▷ Marks node as dead
25:         end if
26:     end foreach
27:     return (⊥, ∅)
28: end function
```



(a) vGraph for ϕ with first acceptable cut highlighted

(b) hGraph for ϕ

Figure 3-7: Vertical and horizontal graphs for the formula $\phi = ((a \vee b) \wedge (c \vee (d \wedge (f \vee (e \wedge g)))) \wedge h)$

we search for the second one starting with the root nodes containing the literals a and b. Both of these nodes are leading only to nodes in the first cut or to dead nodes. We chose b because it is not part of the cut itself. From b, the only node we can chose is the one containing d. From there, we once again have to choose between to nodes leading to nodes in the cut, so we choose the one which is not part of the cut itself. Finally we pick the last node, containing the literal g. Of the three acceptable cut we could create, this one is the least-overlapping. Algorithm 4 details the process.

3.3 Practical considerations

We have described the theoretical aspects of Boolean constraint propagation and conflict detection on graphs. Some details need to be mentioned in order to get a fuller understanding of how this can be implemented in practice.

- The number of edges in vGraphs and hGraphs can grow quadratically with the size of the formula. This problem was already mentioned in [21]. We can observe that these graphs have certain interesting properties; for example, if a node v_1 has outgoing edges to nodes $w_i, \ldots w_j$, then all other nodes v_k which have outgoing edges to one of the same nodes $w_l, i \leq l \leq j$ must also have

Algorithm 4 Search for a least overlapping cut

```
1: function CutSearch2(G_V(φ), A, C_1)                          ▷ C_1 is the first cut
2:     (V, E, R, L, c) ← G_V(φ)
3:     ∀v_i ∈ V, leading(v_i) ← ⊥
4:     ∀v_i ∈ C_1, leading(v_i) ← ⊤
5:     return FindLeastOverlapping(R, ∅)
6: end function
7: function FindLeastOverlapping(V, p, C_1)
8:     (ok, p_2) ← FindNonOverlapping(V, p, C_1)
9:     if ok = ⊤ then
10:        return p_2
11:    else
12:        if ∃v ∈ V(leading(v) ∧ v ∉ C_1) then
13:            return FindLeastOverlapping({w|(v, w) ∈ E}, p → v, C_1)
14:        else
15:            v ← C_1 ∩ V                          ▷ has to exist and be a singleton
16:            return FindLeastOverlapping({w|(v, w) ∈ E}, p → v, C_1)
17:        end if
18:    end if
19:    return (⊥, ∅)
20: end function
21: function FindNonOverlapping(V, p, C_1)
22:    foreach v ∈ V do
23:        if marked(v) = ⊤ ∨ c(v) is false in A ∨ leading(v) = ⊤ then
24:            continue
25:        end if
26:        S ← {w|(v, w) ∈ E}
27:        if S = ∅ then
28:            return (⊤, p)
29:        end if
30:        (ok, p_2) ← FindNonOverlapping(S, p → v)
31:        if ok = ⊤ then
32:            return (⊤, p_2)
33:        else
34:            if ∃s ∈ Sleading(s) = ⊤ then
35:                leading(v) ← ⊤
36:            else
37:                marked(v) ← ⊤
38:            end if
39:        end if
40:    end foreach
41:    return (⊥, ∅)
42: end function
```

outgoing edges to all the other ones. The same is true for incoming edges, and hold both in the vertical and horizontal graphs. Using that property, we store edges as a relation between *lists* of nodes rather than between nodes. Any node list has exactly one successor list and one predecessor list, except for the roots and leaves. Besides, any node is in exactly one node list. The similar solution proposed in [21] is to use *hyperedges* between sets of nodes. It is not difficult to see how these hyperedges can be implemented with lists.

- We mentioned that we saved the status of "dead" nodes between the search for the first and the second cut. In fact, we also save that information during the global DPLL search, as new assignments can only make more nodes dead. We maintain the last decision level when a particular graph had its "dead-list" updated, and we only revoke that list if the DPLL search backtracks beyond that decision level.

- Another important observation is that when cuts start to overlap, they can never be non-overlapping before backtracking returns the assignment status for the exact same reason. In order to avoid continuously searching for non-overlapping cuts when such a thing wouldn't be possible, we also store the decision level at which their intersection stopped being empty. We then don't search for new cuts as long they both stay acceptable.

3.4 Use as part of NC(T)

The modular architecture we propose for non-clausal SMT solvers, NC(T), is not based solely on the graphical representation of formulas. We do believe that the work that has been done on clausal solvers is remarkable and needs to be leveraged. We have already seen that the constraints extracted from the learning process during conflict detection are best expressed as clauses, and this alone is a sufficient motivation to integrate the non-clausal approach together with a clausal one. Another incentive is that many existing benchmarks are only available in CNF. Whether this is a con-

sequence of the dominance of clausal solvers or simply a more appropriate encoding for some problem families is unknown to us. The fact remains that using techniques designed for clausal representations on purely clausal problems is obviously the right decision. Finally, we noticed when we converted non-clausal examples to NNF that about 18% of them turned out to contain some clausal constraints.

The approach we have chosen is to split the formula on top-level conjunctions after its conversion to NNF. Clausal constraints are then treated in the conventional way, using the two-watched-literal scheme, while non-clausal ones are watched by cuts using the techniques we have described. Splitting top-level conjuncts has the additional advantage that, while we have to maintain different cuts for each non-clausal constraint, these cuts are much shorter, and when one part of them becomes unacceptable, the search for a new is considerably shortened, as the traversed vGraph is shallower. The approach detailed in [19] is apparently similar[3], although we were only made aware of this after our implementation was completed.

Whenever an implication or a conflict is found in one of the vGraph-hGraph pairs corresponding to a non-clausal constraint, the implicant clause is computed by traversing the hGraph, and we then add it to the set of inspected clauses. When we compute the implications or detect the conflicts arising from a new literal assertion, we always start by inspecting the clauses, as this is presumably faster, given that the algorithms working on graphs are less mature. This presents the advantage that conflicts or implications which were detected once in the graphs do not need to be detected again, should they arise again after backtracking, for instance. In the case of conflicts, we save the time of realizing that there is no acceptable cut anymore, and in the case of implications, we save the time of searching through the horizontal graph for an implicant clause. Finally, it is useful to record the implicants as clauses to be able to easily recover them when they are needed to build the implication graph.

The interaction with the theory is essentially not modified from DPLL(T): theory implications are computed each time a new literal is asserted or when Boolean consequences are propagated by the clausal and non-clausal frameworks. This is

[3]Himanshu Jain, private communications

implemented as a loop which terminates when no new theory-generated, clausal, or non-clausal consequences are found.

Chapter 4 gives more details about fSTP, our implementation of NC(T) for $T^{=}_{\mathsf{UF}}$.

3.5 Possible improvements

Even before we analyze our results (in Chapter 5), we can notice that there is room for improvement in the theoretical framework. Here are a few points that we believe should be addressed or can be improved:

- Not all cuts have the same size. The formula displayed in Figure 3-7 for instance has vertical paths of length 3, 4 and 5. While the algorithms that we have presented to find cuts do not try to minimize their sizes, it seems reasonable to believe this could improve the efficiency of conflict detection, the rationale being that smalled cuts will remain acceptable longer. Rather than trying to minimize the number of nodes, one should in fact attempt to find cuts with a small number of different literals. This second option is more difficult to put into practice though, as one cannot simply order the list of successor nodes. Another possibility to consider would be to avoid cuts with literals and their negations, as these are trivially bound to eventually become unacceptable.

- A somehow similar issue comes with the generation of implicant clauses. We do not currently attempt to minimize the size of the clause we produce. It is not obvious that minimizing the number of literals would necessary be the right choice here, though. For instance, it could be beneficial when we compute clauses that generated conflicts to try and find one with the *minimal maximum* decision level among its literals. This would allow more powerful backtracking in the global search.

Chapter 4

Implementation

This chapter gives an overview of fSTP, our implementation of the NC(T) scheme for $T_{\mathsf{UF}}^{=}$. Note that we do not describe the implementation of the theory solver, as it is entirely based on existing literature [16, 27, 28].

fSTP is written in C++. The whole system amounts to approximately 8000 lines of code, excluding blank lines and comments. The theory solver and the Boolean solver account for approximately 3500 lines each, with the remaining 1000 consisting of data structures used in both parts. The Boolean solver is written using class templates only, meaning that it could be instantiated for a new theory solver, in effect porting the versatility of DPLL(T)/NC(T) to the actual implementation.

Figure 4-1 shows an overview of the system. The phases depicted are standard for a solver:

- At the *entry point*, arguments to fSTP are parsed. Flags can be set to enable for instance the output of a satisfying assignment at the end of the search, or

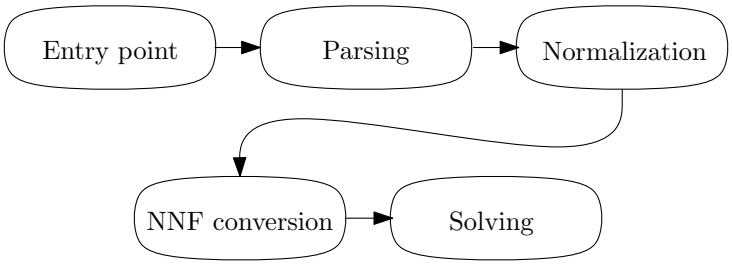


Figure 4-1: fSTP system overview

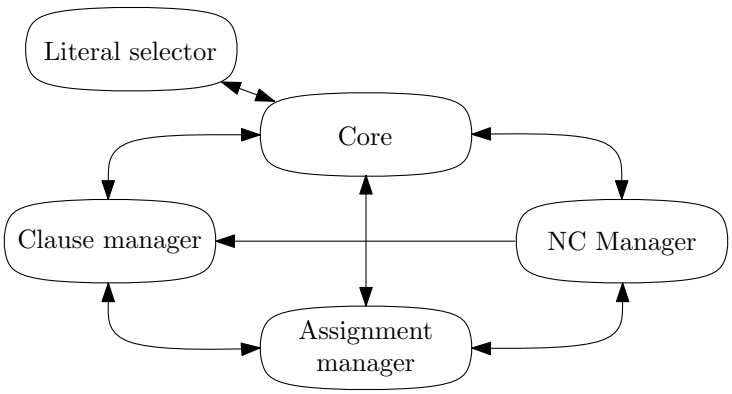


Figure 4-2: Overview of the NC(T) solver

to specify a random seed manually. The name of the file containing the input formula is also read.

- The benchmark file is *parsed*, using a standard Flex/Bison combination, and conforming to the standard defined by the SMT-LIB [31]. The abstract syntax tree of the formula is built, using hash tables to share common nodes.

- The literals in the formula are then *normalized*, a phase where for instance it is established that $a = b$ and $b = a$ describe the same literal.

- The formula is then *converted* to NNF, following the process described in Chapter 3.

- Finally, the formula is passed to the NC(T) solver, along with the parameters specified at the entry point. The next section presents the components of the solver and their interaction.

4.1 Organization of the NC(T) solver

Figure 4-2 shows the different components of the NC(T) solver, using arrows to denote the circulation of information between them. We now describe each of them.

4.1.1 Assignment manager

The *assignment manager* maintains at all times the assignment under consideration. For each assigned literal, it also records the reason of the assignment, as follows. For literals implied by Boolean constraint propagation, it stores a pointer to the clause which triggered the implication. For theory implications, it simply sets a flag. If the actual implicants end up being required, for instance during conflict analysis, the theory solver can then be queried accordingly.

There are two other types of assigned literals, which neither require the storage of extra information: asserted literals, and literals which are directly implied by the structure of the formula —for instance because it contains a top-level conjunction with some single literals. All the other components of the solver refer to the assignment manager to determine the status of literals, for instance when examining watches in clauses or cuts in the non-clausal manager. They also communicate new implications directly to the assignment manager.

4.1.2 NC manager and clause manager

The *NC manager* does Boolean constraint propagation and conflict detection on the set of non-clausal constraints, as described in Chapter 3. Whenever it discovers a new implication, it computes an implicant clause and communicates it to the clause manager.

The *clause manager* performs BCP and conflict detection on the clausal constraints using the two-watched-literal scheme, as described in Chapter 2. It differentiates between clauses which are part of the original formula and learnt clauses, as the latter could be soundly removed after some time to alleviate the memory consumption, while the former cannot. We have however not yet implemented any form of clause set simplification. We believe this lack to be a major problem in our current implementation (see Chapter 5).

4.1.3 Literal selector

The *literal selector* simply maintains the list of unassigned literals constantly sorted by their VSIDS score (see Chapter 2). This is implemented as follows: the data structure used for the list is an STL linked list. When the score of a literal is incremented, it is removed from the list and reinserted at the correct new position, in a way similar to a *bubble sort* algorithm. When a literal gets assigned, either because it was asserted or because it was found to be implied, it is removed from the list as well as its negation, and inserted into a stack containing all assigned literals and their negations. The size of that stack is stored for each decision level. At backtracking, the proper number of elements is popped from the stack and re-inserted into the linked list using a *merge sort*.

When requested to return the literal with the highest score, the literal selector simply starts from the beginning of the linked list. If more than one literals have the same maximum score, one of them is picked randomly.

4.1.4 Core

We now turn to the *core* of the NC(T) solver, which implements the main loop of the search algorithm. The operations taking place in the core are depicted in Figure 4-3.

- The *preprocessing* step splits the main formula into single literals, clausal constraints and non-clausal constraints. Single literals are immediately communicated to the assignment manager, clauses are sent to the clause manager and non-clausal constraints to the NC manager.
- The first step of the search loop is *literal selection*, which is essentially done by a simple query to the literal selector.
- After the selection, the new literal is passed to the theory solver, which either returns new implications or reports itself as being in a conflicting state.

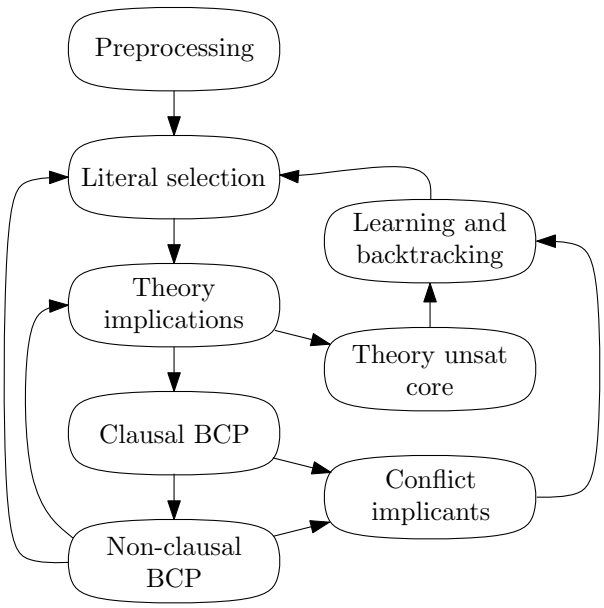


Figure 4-3: Overview of the solving process

- If no conflict was found, the newly selected literal *and* the theory implications are passed to the clause manager which performs BCP and conflict detection using these new facts. Just like the theory solver, this can result in potential new implications or a conflicting state.

- If still no conflict was found, the set comprising the asserted literal, the theory implications and the Boolean implications is passed to the NC manager which analyzes the non-clausal constraints in the light of the new assignments and itself returns potential new consequences or establish that a conflict has occurred.

- If any new implications were found and if the solver is not in a conflicting state, the feedback loop returns to the theory solver, then to the clause manager and finally to the NC manager until no new implication is found. The solver then increments the decision level, picks a new literal and reiterates the process.

- Whenever a conflict arises, an implicant clause is computed, either by querying the theory solver in the case of a theory conflict, of by traversing the implication graph otherwise. The clause in question is then transformed to include a first unique implicant, and the solver backtracks to the appropriate decision level

(see Chapter 2).

The NC(T) solver returns sat if during literal selection it is found that all literals have been assigned a value, or unsat if during conflict analysis it is found that the conflict is not resolvable.

Chapter 5

Experimental Results

We tested fSTP, our implementation of the NC(T) framework for $T^{=}_{\mathsf{UF}}$ on a large set of examples available through SMT-LIB [31]. We split the test cases into two classes, separating the problems for which the NNF conversion yields a formula in CNF from the other, non-clausal ones.

Applying the NC(T) approach on the examples in CNF is essentially similar to using DPLL(T) alone, as the non-clausal conflict detection and Boolean constraint propagation parts are never used. This class, comprising approximately 600 test cases, thus provides us with a reasonably accurate way of measuring how our DPLL(T) implementation compares with existing tools.

The number of available non-clausal examples is significantly larger. To keep our measurement procedure within reasonable time constraints, we cut it down to a figure similar to the number of clausal ones. We did this by simply ordering the examples alphabetically and picking every tenth of them. A tacit naming convention in the SMT-LIB seems to be sufficiently well followed, implying that this simple technique allowed us to gather a subset of examples representative of the complete collection in terms of families and sizes. Table 5.1 shows the breakdown of selected benchmarks by classes and problem families. For the non-clausal instances, the total available number is shown in parentheses.

We compared fSTP to BarcelogicTools 1.2 [26], which was the latest available version when we started working on NC(T). While it is not the best solver available

	Clausal	Non-clausal
NEQ	48	(none)
PEQ	48	(none)
SEQ	56	(none)
QG_classification	430	548 (5526)
QG/loops6	16	42 (432)
eq_diamond	1	9 (99)
Total	599	599 (5958)

Table 5.1: Breakdown of benchmarks by classes and families

for $T^{=}_{\mathsf{UF}}$, we felt the comparison would be adequate considering our implementation of the theory solver is entirely based on theirs. We compared both these implementations by solving purely conjunctive formulas, in which the Boolean structure played no part in the solving process, and found them to be roughly on par. This was not the case with Yices [14], for instance. We acknowledge fSTP is not mature enough to be competing among the best, considering it has been developed from scratch over a period of only a few months. Nevertheless, the comparison against a DPLL(T)-based tool can still give some insight about the potential of the NC(T) approach.

5.1 Running times

We ran all our benchmarks with both solvers on a 2 quad-core Xeon 2.66GHz with 16GB of RAM, running Debian GNU/Linux (64bit). Neither solver uses parallelization though, and we observed comparable results on a standard desktop machine. Table 5.2 shows the time required to solve each family. (Note that much higher number of timeouts using fSTP renders the comparison of the running time meaningless.) Table 5.3 shows the timings for the subset of benchmarks that were solved in less than a minute by both solvers. Note that for fSTP these times are identical, since there are no examples solved within the time limit by fSTP but not by BarcelogicTools.

The time measurements and the difference in the number of timeouts already show that the performance of fSTP is rather poor compared to the chosen competing tool. Figure 5-1 and Figure 5-2 show all clausal and non-clausal benchmarks respectively,

		bclt	fSTP
Clausal	NEQ	952 (6)	178 (38)
	PEQ	2652 (23)	6.0 (47)
	SEQ	691 (6)	3.6 (45)
	QG_classification	2735 (15)	870 (154)
	QG/loops6	1.3 (0)	0.9 (5)
	eq_diamond	0.0 (0)	0.0 (0)
Non-clausal	QG_classification	519 (4)	1441 (85)
	QG/loops6	21 (0)	58 (5)
	eq_diamond	41 (6)	0.3 (7)

Table 5.2: Comparison of running times of fSTP and BarcelogicTools, by benchmark family. All times are indicated in seconds. The timeout is set to 60 seconds. The numbers in parentheses indicate the number of timeouts. The solving time does not include timeouts.

		N	bclt	fSTP
Clausal	NEQ	10	3.3	178
	PEQ	1	0.2	6.0
	SEQ	11	1.4	3.6
	QG_classification	276	338	870
	QG/loops6	11	0.8	0.9
	eq_diamond	1	0.0	0.0
Non-clausal	QG_classification	463	200	1441
	QG/loops6	36	16	58
	eq_diamond	2	0.0	0.3

Table 5.3: Comparison of running times of fSTP and BarcelogicTools, including only benchmarks where both solvers finish on time. N indicates the number of examples considered in each family.

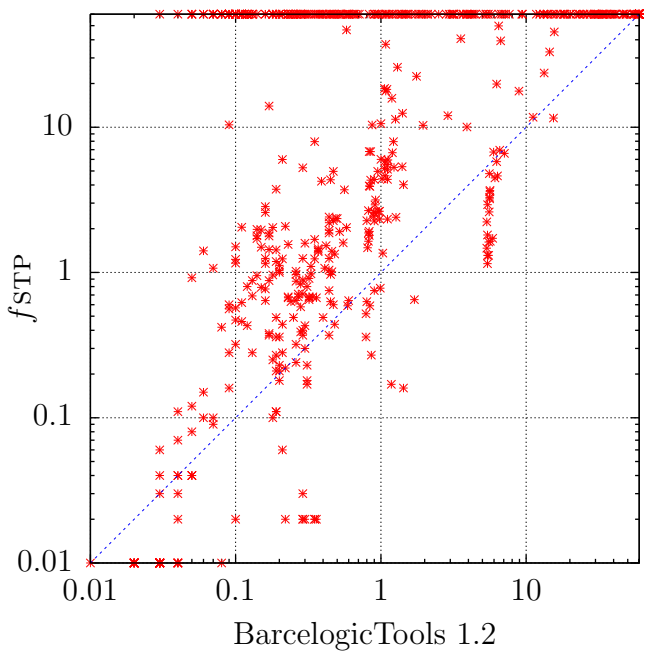


Figure 5-1: Running times for clausal benchmarks, compared to BarcelogicTools 1.2

represented as points on a scatter plot where the x axis indicates the time required to solve them using BarcelogicTools, and the y axis using f_{STP}. Points above the $x = y$ diagonal therefore represent instances solved faster using BarcelogicTools.

5.1.1 Clausal benchmarks

We attribute the large difference in performance on clausal instances to the lack of tuning and the missing features of our DPLL(T) implementation. While this may sound obvious, considering that most DPLL(T)-based solvers are based on similar, well-known features, here are the main points that we believe we should address in future improvements to the DPLL engine of our tool:

- The current single most important issue seems to be that we do not do any sort of minimization on the set of learnt clauses and as a result, let it grow to unreasonable proportions during the search, significantly affecting the rate of decisions per second on problems which take a long time to solve. We do not delete clauses, nor do we attempt to reduce or merge existing ones. The former

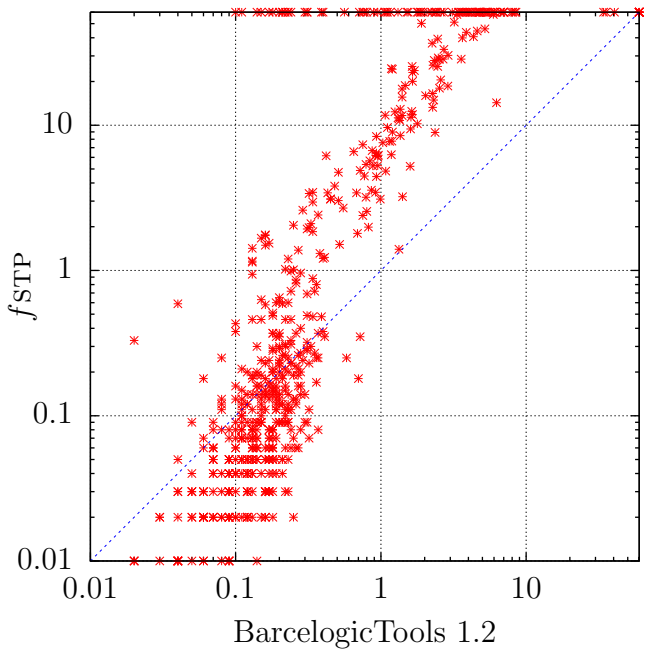


Figure 5-2: Running times for non-clausal benchmarks, compared to BarcelogicTools 1.2

technique is a common feature of SAT and SMT solvers [25, 29] while the latter has been shown to be efficient on some hard SAT instances [33]. Furthermore, Ryan [32] links the performance of the VSIDS heuristic to clause resolution, postulating that its real power comes from the resolution-prone clauses that are generated when in use, rather than simply from the fact that it drives the search towards literals that satisfy the most recently learnt clauses, as originally suggested in [25].

- We do not treat binary clause differently that longer clauses. This is known to be inefficient and better techniques exist [32].

- We have not yet fine-tuned the VSIDS heuristic. We have observed that the solving time for some instances could immensely vary when we ran the search with different random seeds. We postulate that this is because we do not add transient randomness to the literal selection procedure after restarts as suggested in [25], however this requires further investigation.

- We have not experimented sufficiently with the restart sequences, which can play a decisive part in the efficiency of search-based solvers [18]. We use a policy based on [24] as do other solvers [30], although we have gathered no evidence that would determine whether this is the best choice for us.

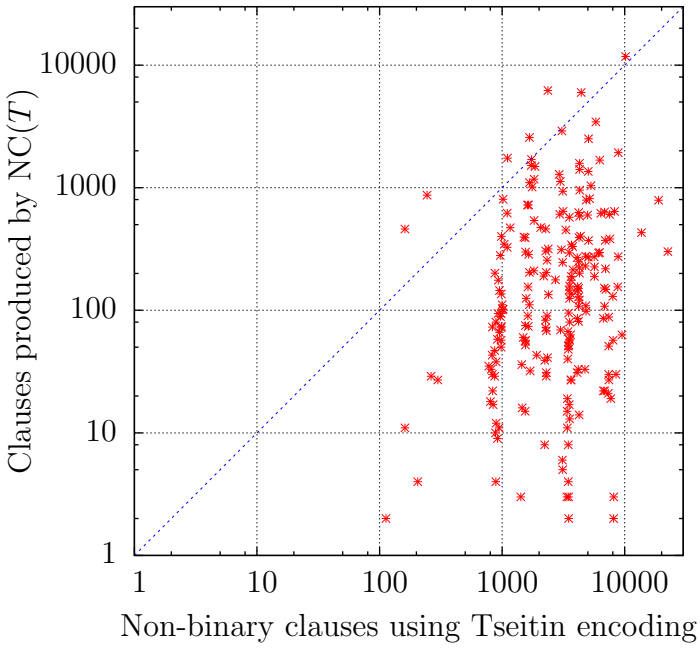
5.1.2 Non-clausal benchmarks

Considering the low performance of our DPLL(T) implementation, a direct comparison of our results on non-clausal examples with BarcelogicTools makes little sense. Indeed, we have already stressed the capital importance that the DPLL(T) part plays in the broader NC(T) framework; the non-clausal reasoning creates a substantial number of clauses which are then treated by the DPLL engine, as it is presumably more efficient. It is therefore reasonable to link the observed performance on non-clausal examples at least partially to the performance on clausal ones.

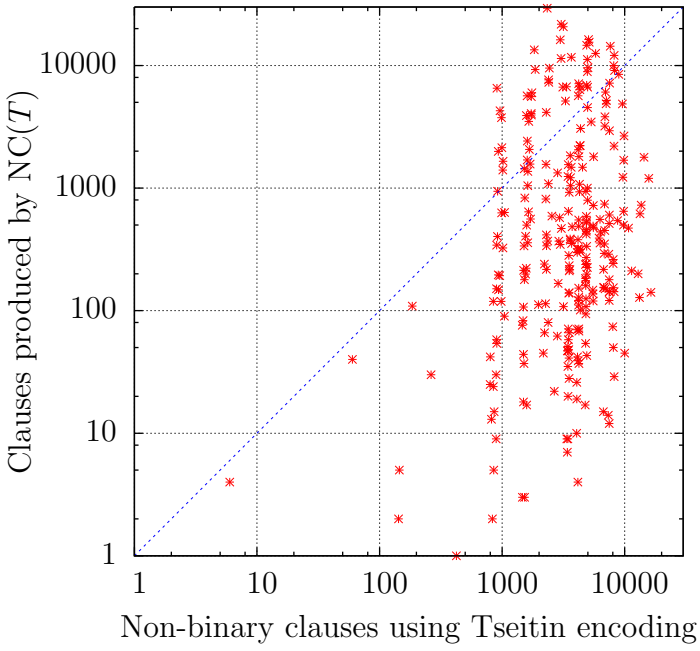
To get a better sense of the potential of the NC(T) approach, we did the following experiment: we converted every problem in the non-clausal set to CNF using the straightforward Tseitin encoding [35]. We then compared the number of clauses generated with this method with the number of clauses generated by NC(T) during the search. Measurements confirm the intuition that the size of the clause set should be smaller in the latter cases, as clauses are extracted by the non-clausal reasoning part only when they play a role in the search. Figure 5-3 shows the comparison between these figures. We can observe that the ratio is in general even more in favor of NC(T) for `sat` instances. This is expected, as the search in this case does not go through every possible assignment and therefore triggers less conflicts and implication.

We should note that when counting the number of clauses generated using the Tseitin encoding, we excluded binary clauses which would dramatically and unrealistically increase the figure. Indeed, a large number of binary clauses results from that encoding, but as previously mentioned, state-of-the-art solvers handle them very efficiently.

Figure 5-4 shows the evolution of the size of the clause set as the search progresses, giving an indication on the proportion of conflict and NC(T)-derived clauses. We have



(a) sat instances



(b) unsat instances

Figure 5-3: Clauses generated by fSTP using NC(T) compared to equivalent Tseitin encoding

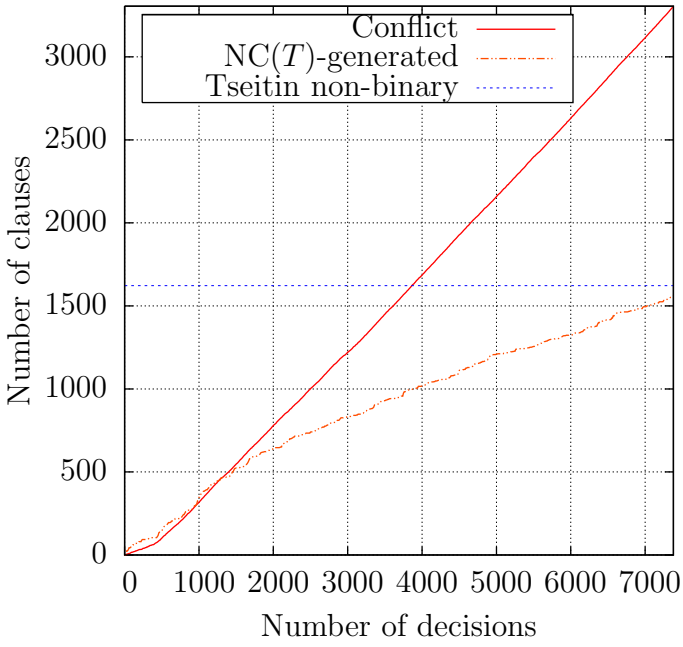


Figure 5-4: Comparison of conflict and NC(T)-generated clauses during the solving process for an unsat formula. The number of non-binary Tseitin clauses after CNF conversion is given for reference.

found this pattern to be typical: while the number of conflict clauses grows linearly with the number of decisions, the rate of clauses added from implications found in the NC(T) part decreases with time. We believe this constitutes evidence that NC(T) can be a worthy addition to DPLL(T), as it works as an *incremental translation* to CNF, progressively converting a non-clausal problem into one that can be handled by efficient solvers, rather than translating it entirely upfront.

Chapter 6

Related work

We have already referred to [21], which constitutes research in progress and was generously shared with us. A complete account of Jain's work is about to be published as his PhD thesis [19]. Jain et al. already published some of their earlier work in [20], where they present a graph-based non-clausal SAT solver.

Rather than using a DPLL-like recursive search, the approach they propose there consists in searching directly for a vertical path, while collecting conflict clauses from the horizontal graph. The number of vertical paths to be explored is further pruned down by the learning of *local* conflict clauses which are attached to nodes and constrains the paths going through them.

[34] presents another attempt at a non-clausal SAT solver. The approach chosen by Thiffault et al. doesn't require conversion to NNF and works by adapting the two-watched literal idea to arbitrary formulas represented as directed acyclic graphs, by watching literals in conjunctions and disjunctions and propagating relevant information to both parent and child nodes.

Problems arising from common computer-aided design tasks generally have the structure of Boolean circuits. Standard DPLL-based SAT solvers have proved successful in these tasks. However, there are also advantages to working directly with formulas encoded as directed acyclic graphs containing only negations and conjunctions [7, 22].

[2] presents a generalization of the DPLL algorithm for non-clausal formulas. The

search space is pruned down by adapting the pure literal and unit literal rules of DPLL to the non-clausal case. It is also mentioned that the procedure can be applied to checking validity of formulas in quantifier-free first-order logics, provided that a decision procedure exists for testing the satisfiability of a partial assignment in these logics. The principle described is the same as the one that led to the DPLL(T) scheme, although no mention of communication from the theory decision procedure to the Boolean solver is made. As far as we know, this technique has only been evaluated on random test cases.

Overall, we believe that NC(T) presents a new point in the design space of efficient algorithms for SAT and SMT.

Chapter 7

Conclusion

We have presented NC(T), our proposal for an extension of the DPLL(T) scheme to include reasoning on non-clausal constraints, as well as our implementation of NC(T) for the quantifier-free theory of uninterpreted function symbols and predicates with equality, and have shown some experimental results.

Our overall impression is that the NC(T) scheme has not yet shown its full potential. We acknowledge that our practical results do not place fSTP among the existing state-of-the-art SMT solvers for $T^{=}_{\mathsf{UF}}$, and we attribute this largely to an immature implementation, in particular of the DPLL(T) scheme itself. The points which are particularly affected by the lack of maturity are the management of learnt clauses and the untuned state of the various heuristics.

We have shown however how our non-clausal extension to DPLL(T) can act as an alternative to upfront translation to conjunctive normal form, by providing in effect an *incremental translation* as the search progresses. We have demonstrated that the number of clauses generated this way is often substantially smaller than the number of equivalent non-binary clauses resulting from Tseitin encoding, particularly on `sat` instances where the search doesn't traverse all possible assignments.

Our future work will focus on improving our implementation, in an attempt to bring the DPLL part to a competitive level, thus enabling us to then verify our hypothesis that NC(T) can be a beneficial addition to it. We have already mentioned in the previous chapter what we believe to be the main bottlenecks and will focus on

these points.

We also believe additional research needs to be done on literal selection in the context of SMT solving. Existing implementations seem to rely on the assumption that techniques which have been developed for SAT will work equally well in the context of DPLL(T). While no data seems to currently contradict that hypothesis, we postulate that literal selection could be improved by implying the theory solver in decisions and plan to investigate that idea.

Bibliography

[1] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Springer (Kluwer), 2nd edition, 2002. 26, 30

[2] A. Armando and E. Giunchiglia. Embedding Complex Decision Procedures Inside an Interactive Theorem Prover. *Annals of Mathematics and Artificial Intelligence*, 8(3):475–502, 1993. 57

[3] C. Barrett, L. de Moura, and A. Stump. Design and Results of the 1st Satisfiability Modulo Theories Competition (SMT-COMP 2005). *Journal of Automated Reasoning*, 35(4):373–390, 2005. 13

[4] C. Barrett and C. Tinelli. CVC3. *Proceedings of the 19th International Conference on Computer Aided Verification, CAV*, pages 298–302, July 2007. 13

[5] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking Using SAT Procedures Instead of BDDs. *Design Automation Conference*, pages 317–320, 1999. 9

[6] P. Bjesse, T. Leonard, and A. Mokkedem. Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers. *Lecture notes in computer science*, pages 454–464, 2001. 9

[7] R. Brummayer and A. Biere. Local two-level and-inverter graph minimization without blowup. *Proc. MEMICS*, 6, 2006. 57

[8] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automati-

cally Generating Inputs of Death. *Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, 2006. 9

[9] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001. 9

[10] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, 1962. 14

[11] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960. 14

[12] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008. 13

[13] P. Downey, R. Sethi, and R. Tarjan. Variations on the Common Subexpression Problem. *Journal of the ACM (JACM)*, 27(4):758–771, 1980. 21

[14] B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI, 2006. Tool paper at `http://yices.csl.sri.com/tool-paper.pdf`. 13, 50

[15] N. Eén and N. Sörensson. An Extensible SAT-solver. *SAT*, 2919:502–518, 2003. 13, 14, 19

[16] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. *Proceedings of the 16th International Conference on Computer Aided Verification, CAV*, 4:175–188, 2004. 3, 10, 11, 20, 21, 43

[17] E. Goldberg and Y. Novikov. BerkMin: A fast and robust Sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007. 13

[18] J. Huang. The Effect of Restarts on the Efficiency of Clause Learning. *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI07)*, pages 2318–2323, 2007. 19, 54

[19] H. Jain. *Verification using Satisfiability Checking, Predicate Abstraction, and Craig Interpolation*. PhD thesis, Carnegie Mellon University, School of Computer Science, 2008. 3, 10, 11, 31, 34, 40, 57

[20] H. Jain, C. Bartzis, and E. Clarke. Satisfiability Checking of Non-clausal Formulas using General Matings. *9th International Conference on Theory and Applications of Satisfiability Testing*, 2006. 10, 27, 28, 57

[21] H. Jain and E. Clarke. Applying DPLL algorithm to the Graph Based Representations of Non-Clausal Formulas (Unpublished manuscript), 2008. 3, 10, 11, 31, 34, 37, 39, 57

[22] A. Kuehlmann, M. Ganai, and V. Paruthi. Circuit-based Boolean Reasoning. *38th Design Automation Conference (DAC01)*, pages 232–237, 2001. 57

[23] S. Lerner, T. Millstein, and C. Chambers. Automatically Proving the Correctness of Compiler Optimizations. *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 220–231, 2003. 9

[24] M. Luby, A. Sinclair, and D. Zuckerman. Optimal Speedup of Las Vegas Algorithms. *Theory and Computing Systems, 1993, Proceedings of the 2nd Israel Symposium on the*, pages 128–133, 1993. 54

[25] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. *Proceedings of the 38th conference on Design automation*, pages 530–535, 2001. 13, 14, 15, 19, 31, 53

[26] R. Nieuwenhuis and A. Oliveras. Decision Procedures for SAT, SAT Modulo Theories and Beyond. The BarcelogicTools. *LPAR*, pages 23–46, 2005. 13, 22, 49

[27] R. Nieuwenhuis and A. Oliveras. Proof-Producing Congruence Closure. *Term Rewriting and Applications: 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005: Proceedings*, 2005. 11, 43

[28] R. Nieuwenhuis and A. Oliveras. Fast Congruence Closure and Extensions. *Information and Computation*, 205(4):557–580, 2007. 11, 21, 22, 43

[29] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *Journal of the ACM (JACM)*, 53(6):937–977, 2006. 3, 10, 20, 23, 53

[30] K. Pipatsrisawat and A. Darwiche. RSat 2.0: SAT Solver Description. *Solver description, SAT competition*, 2007. 54

[31] S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.smt-lib.org`. 44, 49

[32] L. Ryan. Efficient Algorithms for Clause-Learning SAT Solvers. Master's thesis, Simon Fraser University, 2004. 19, 53

[33] N. Sörensson and N. Eén. Minisat v1.13 - A Sat Solver with Conflict-Clause Minimization. *SAT 2005*, 2005. 53

[34] C. Thiffault, F. Bacchus, and T. Walsh. Solving Non-Clausal Formulas with DPLL Search. *SAT*, 2004. 10, 57

[35] G. Tseitin. On the Complexity of Derivation in Propositional Calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 2:115–125, 1968. 10, 13, 54

[36] M. Velev and R. Bryant. Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, 2003. 9

[37] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. *International Conference on Computer-Aided Design (ICCAD01)*, pages 279–285, 2001. 18, 19