Nenofar: A Negation Normal Form SMT Solver
Combining Non-Clausal SAT Approaches with Theories

Philippe Suter[1], Vijay Ganesh[2], Viktor Kuncak[1⋆]

[1] EPFL, Switzerland
[2] MIT, USA

Abstract. We describe an implementation of a solver for satisfiability modulo theories. Unlike DPLL(T), our solver avoids the translation to CNF. Instead, it uses a new approach, NC(T), which simultaneously handles clauses as well as formulas in negation normal form. We implemented our approach in a system called Nenofar. Our experiments show that, using the same solver engine, the non-clausal approach improves the performance compared to several translations to CNF. Nenofar source code and benchmarks are available on the web for further experiments.

1 Introduction

Modern satisfiability modulo theories (SMT) provers [1, 2] are often based on DPLL(T) [3] and work with clausal representation of formulas. In contrast, formulas arising in applications are often in non-clausal form. For example, the SMT-LIB [4] benchmark suite contains non-clausal benchmarks. In software verification, verification condition generation techniques such as [5, 6] convert loop-free code into polynomially sized formulas whose syntax tree contains alternation of disjunctions and conjunctions, and are therefore also not in clausal form.

A standard approach to bridge the gap between non-clausal problems and clausal techniques is to initially convert the entire formula into the clausal form, using one of the several proposed encodings [7, 8, 9]. These encodings, however, introduce additional variables, and may obscure the syntactic structure of the original formula. Researchers have therefore developed SAT solvers that work on non-clausal representations of formulas, avoiding the need for such translations [10, 11]. Building on this work, our goal is to apply non-clausal techniques to SMT, and to experimentally explore the potential of this approach.

This paper presents Nenofar, NEgation NOrmal Form Automated Reasoner, a prover that decides satisfiability of quantifier-free first-order logic with equality by working directly on negation-normal form (NNF) of formulas. Nenofar makes it easy to compare different approaches to non-clausal formulas. In addition to natively handling NNF, it also implements three translations to conjunctive

⋆ This research was supported in part by the Swiss National Science Foundation Grant "Precise and Scalable Analyses for Reliable Software".

normal form (CNF). Our results show that using NNF is in most cases faster than doing CNF translation.

In the rest of this paper we describe the approach implemented in Nenofar and present our experimental results. More details about the underlying algorithm, NC(T), are in [12]. We believe the approach is promising and useful to study non-clausal reasoning. The tool and the examples from our experiments are available at:

http://lara.epfl.ch/dokuwiki/nenofar

This paper makes the following contributions:

1. We present the first combination of reasoning on theories and non-clausal boolean reasoning in a schema we call NC(T), an extension of DPLL(T).
2. Our modifications to the existing non-clausal framework increase its interaction with the clausal reasoner thus yielding a system which effectively works by incrementally translating to CNF.
3. Finally, we report on the evaluation of our method compared to upfront translations to CNF.

2 Example

This section gives an informal overview of non-clausal reasoning through an example. We describe the techniques involved in more detail in Section 3.

Consider the following non-clausal formula:

$$\begin{aligned}&((f(a) \neq f(b) \vee b = c) \wedge a = b \wedge a \neq c) \vee \\ &(b = f(b) \wedge (b = f(a) \vee (a = b \wedge b \neq f(a))))\end{aligned} \tag{1}$$

To check the satisfiability of (1), a DPLL(T) solver would typically convert the formula to CNF first, introducing fresh boolean variables to avoid an exponential increase in the size of the formula (we describe such encodings in Section 5).

Rather than working on CNF, Nenofar performs reasoning on NNF constraints. A formula is in NNF if it contains only the boolean operators $\wedge$, $\vee$ and $\neg$, and if $\neg$ only appears directly in front of atomic subformulas. Any formula, and in particular any quantifier-free formula, can be efficiently transformed into a logically equivalent formula in NNF of the same structure by applying the rules given in Figure 1. Note that (1) is already in NNF.

Just like any other DPLL(T) solver, Nenofar searches for a satisfying assignment by 1) heuristically asserting truth values to literals, 2) computing the effects of such assignments on the boolean structure of the formula as well as their consistency within the theory. If during the search all literals have a truth value assigned such that the conjunction of these assignments is not self-contradictory and the assignment satisfies the boolean structure, then the search concludes and the formula is satisfiable (SAT). If, on the other hand, all possible assignments are exhausted and none could be found to satisfy the formula, then it is declared unsatisfiable (UNSAT).

$$\mathsf{NNF}(\phi) = \begin{cases} \phi & \text{if } \phi \text{ is atomic} \\ \neg\phi_1 & \text{if } \phi \text{ is } \neg\phi_1 \text{ and } \phi_1 \text{ is atomic} \\ \mathsf{NNF}(\phi_1) & \text{if } \phi \text{ is } \neg\neg\phi_1 \\ \mathsf{NNF}(\phi_1) \wedge \ldots \wedge \mathsf{NNF}(\phi_n) & \text{if } \phi \text{ is } \phi_1 \wedge \ldots \wedge \phi_n \\ \mathsf{NNF}(\phi_1) \vee \ldots \vee \mathsf{NNF}(\phi_n) & \text{if } \phi \text{ is } \phi_1 \vee \ldots \vee \phi_n \\ \mathsf{NNF}(\neg\phi_1) \vee \ldots \vee \mathsf{NNF}(\neg\phi_n) & \text{if } \phi \text{ is } \neg(\phi_1 \wedge \ldots \wedge \phi_n) \\ \mathsf{NNF}(\neg\phi_1) \wedge \ldots \wedge \mathsf{NNF}(\neg\phi_n) & \text{if } \phi \text{ is } \neg(\phi_1 \vee \ldots \vee \phi_n) \end{cases}$$

Fig. 1. Rules for the conversion of quantifier-free formulas into negation normal form.

Critical to the applicability of DPLL-like techniques is the ability to derive new assignments from previous ones, thereby limiting the number of literals which remain to be assigned in a given branch of the search tree. Derived assignments have to be logically implied by: either the conjunction of the previous ones, in which case they constitute *theory implications*, or by the truth value of the assigned literals and the boolean structure of the formula, in which case they are *boolean implications*.

Consider for instance the search for an assignment satisfying (1). If at given point in the search the atom $f(a) = f(b)$ is assumed to be true and the atom $b = f(b)$ is assumed to be false, then:

- $f(a) \neq b$ is a theory implication, because $f(a) = f(b) \wedge b \neq f(b) \Rightarrow f(a) \neq b$.
- $a = b$ is a boolean implication, because adding the assertion $a \neq b$ to the assignment immediately makes the formula unsatisfiable.

In Nenofar, similarly to DPLL(T) solvers, theory implications are computed in a separate component, the theory solver. Boolean implications are derived by reasoning on the NNF constraints. Each formula is represented internally as a pair of directed acyclic graphs (DAG): the *vertical graph* and the *horizontal graph* [13,14,11]. Figure 2 shows these two graphs for (1). A *path* in the vertical graph from a node with no predecessor to a node with no successor corresponds to one conjunction of the formula in disjunctive normal form (DNF). Dually, a path in the horizontal graph corresponds to a clause in CNF.

From the properties of horizontal and vertical graphs one can derive procedures to perform *boolean constraint propagation* and *conflict detection* on NNF formulas. The partial boolean assignment under consideration *implies* a literal when there exists a path in the horizontal graph such that all literals are falsified by the assignment except one. In our example, if $a \neq b$ is in the current assignment, then $b = f(a)$ is an implied literal because of the horizontal path $3 \rightarrow 6 \rightarrow 7$.

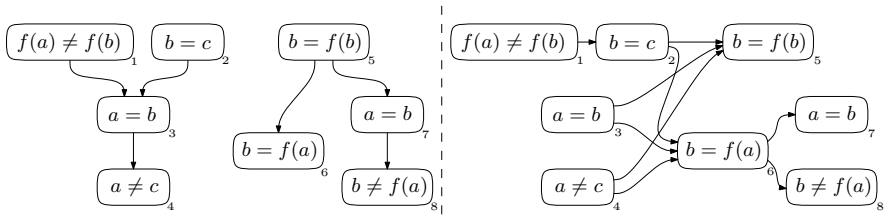

Fig. 2. Vertical and horizontal graphs of (1)

3 Non-Clausal Reasoning in Nenofar

This section describes the algorithms used for reasoning about non-clausal boolean structures in Nenofar. The propositional aspect of the techniques presented here was first described in [14, 11].

3.1 Graph Representations

In [13], 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 shows (1) in this representation.

$$\left[\left[\begin{array}{c} f(a) \neq f(b) \vee b = c \\ a = b \\ a \neq c \end{array}\right] \vee \left[\begin{array}{c} b = f(b) \\ b = f(a) \vee \left[\begin{array}{c} a = b \\ b \neq f(a) \end{array}\right] \end{array}\right]\right]$$

Fig. 3. Two-dimensional representation of (1)

A *vertical path* in this representation is obtained by traversing the diagram from top to bottom, choosing for each traversed disjunction exactly one disjunct to follow. As an example, $(b = c) \rightarrow (a = b) \rightarrow (a \neq c)$ is a vertical path in Fig. 3, and so is $(b = f(b)) \rightarrow (a = b) \rightarrow (b \neq f(a))$. *Horizontal paths* are defined in a similar way, and an example in the same diagram is $(f(a) \neq f(b)) \rightarrow (b = c) \rightarrow (b = f(a)) \rightarrow (a = b)$.

A natural way to represent the sets of paths is by exhibiting a pair of directly acyclic graphs. The graphs corresponding to Figure 3 were shown in Figure 2 in the previous section. We describe these graphs using 5-uples (V, E, R, L, c) where: V is the set of vertices. $|V|$ is equal to the number of literal occurrences

in the formula, multiple occurrences of the same literal being counted multiple times. $E \subset V \times V$ is the set of edges. $R \subseteq V$ is the set of nodes with in-degree 0 and $L \subseteq V$ the set of nodes with out-degree 0. Finally, $c : V \to \mathbb{L}$ is a function mapping each node in the graph to a literal from the formula. For a formula ϕ, we denote its vertical and horizontal graphs as $G_V(\phi) = (V_V, E_V, R_V, L_V, c_V)$ and $G_H(\phi) = (V_H, E_H, R_H, L_H, c_H)$ respectively, and in particular, we have that $V_V = V_H$ and that $\forall v \in V_V \;.\; c_V(v) = c_H(v)$. Algorithm 1, adapted from [14], presents a mechanical procedure to build $G_V(\phi)$ for a formula ϕ. Note that in the presentation, $\wedge$ and $\vee$ are considered to be binary for simplicity, but the adaptation to the n-ary case is straightforward. The construction of the horizontal graph is done is a similar, dual, way. We should note that the process can lead to different pairs of graphs for a given formula, as the operands of $\wedge$ and $\vee$ can be selected in any order. For instance, Figure 4 shows a possible different outcome of the construction for (1). Although the graphs are different, one can easily verify that for a given formula, the set of vertical and horizontal paths are identical, up to permutations in the paths.

Algorithm 1 Vertical graph construction

function $G_V(\phi)$
Require: ϕ is in NNF
  if ϕ is a single literal l **then**
    return $(\{v_i\}, \{v_i\}, \{v_i\}, \emptyset, \{v_i \mapsto l\})$ ▷ where i is a fresh index
  else if ϕ is $\phi_1 \vee \phi_2$ **then**
    $(V_1, E_1, R_1, L_1, c_1) \leftarrow G_V(\phi_1)$
    $(V_2, E_2, R_2, L_2, c_2) \leftarrow G_V(\phi_2)$
    return $(V_1 \cup V_2, E_1 \cup E_2, R_1 \cup R_2, L_1 \cup L_2, c_1 \cup c_2)$
  else if ϕ is $\phi_1 \wedge \phi_2$ **then**
    $(V_1, E_1, R_1, L_1, c_1) \leftarrow G_V(\phi_1)$
    $(V_2, E_2, R_2, L_2, c_2) \leftarrow G_V(\phi_2)$
    return $(V_1 \cup V_2, R_1, L_2, E_1 \cup E_2 \cup (L_1 \times R_2), c_1 \cup c_2)$
  end if
end function

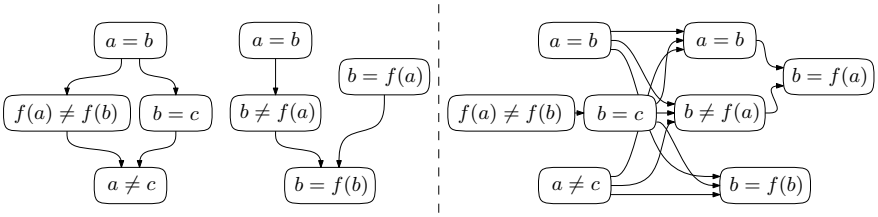

Fig. 4. Alternative vertical and horizontal graphs of (1)

3.2 Detection of Boolean Implications and Conflicts

As suggested in the previous section, the horizontal graph can be used to detect boolean implications resulting from a partial assignment to the literals. Exploring the whole horizontal graph and searching for paths which are almost entirely falsified is too costly, however. One of the main contributions from [11] is an algorithm for efficiently detecting such implications and conflicts while only visiting a small part of the horizontal graph.

We define a *cut* for a pair of vertical and horizontal graphs as a set of nodes such that it contains at least one node from each horizontal path. Jain showed that the set of nodes from any vertical path constitutes a valid cut [11, Appendix C]. Moreover, such cuts are *minimal* in the sense that 1) they contain exactly one literal from each horizontal path, and 2) any cut will always contain at least one vertical path.

We say that a literal l is in a cut if there exists a node v in that cut such that $c_V(v) = l$. Given a partial assignment, we call a cut *acceptable* if none of the literals in it are falsified by the assignment. Whether or not a cut contains a literal and its negation is not a criterion for acceptability as long as that literal remains unassigned. Finally we say that two cuts are *overlapping* if the intersection of their node sets is non-empty. Having common literals is not a sufficient condition, as there could be two different nodes v_1 and v_2 in the first and second cut respectively such that $c_V(v_1) = c_V(v_2)$. Non-overlapping cuts are said to be *disjoint*. Figure 5 shows two disjoint cuts on the vertical and horizontal graphs of (1).

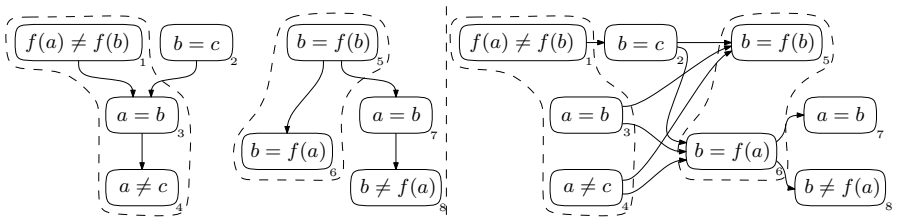

Fig. 5. Two disjoint cuts

The use of cuts on NNF constraints is very similar to the two-watched-literal scheme [15] used in modern DPLL-based solvers working on clauses: the solver tries to maintain two acceptable, disjoint cuts during the search. A failure to find any acceptable cut indicates a boolean conflict in the current assignment, and the inability to find two disjoint cuts indicates boolean implications. The first property follows directly from the construction of the horizontal and vertical graphs: by construction, all vertical paths have exactly one node in common with each horizontal path. Therefore, when a horizontal path is entirely falsified by an assignment, no vertical path can form an acceptable cut. The second property follows from similar arguments: the non-existence of a second acceptable cut

disjoint from the first one implies that there are horizontal paths with only one literal left non-falsified. In particular, this means that if the two overlapping cuts are *minimally overlapping*, then the set of nodes at their intersection contains literals implied by the current assignment. Figure 6 shows an example of such a situation: the literal $b = f(b)$ was assumed to be false. As a consequence, the cuts have $a = b$ and $a \neq c$ as overlapping literals. These two literals are therefore a consequence of the assignment, and one can verify by inspecting the horizontal graph that there are indeed one path for each of them where they are the only non-falsified node (paths $(a = b) \rightarrow (b = f(b))$ and $(a \neq c) \rightarrow (b = f(b))$ respectively).

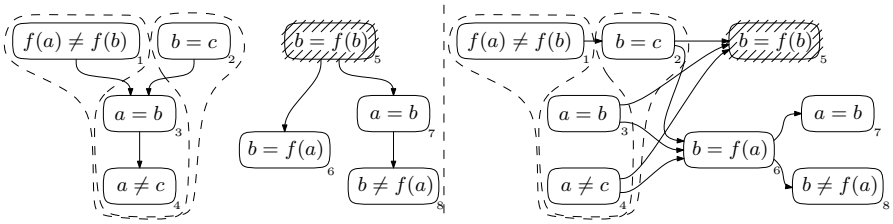

Fig. 6. Minimally overlapping cuts

From a practical point of view, finding the first acceptable cut is essentially a depth-first search on the vertical graph where we mark falsified nodes and nodes leading only to falsified nodes to avoid traversing them multiple times. Algorithm 2 details the process.

The search for a minimally overlapping cut is similar and shown in Algorithm 3: during the second traversal, we mark not only nodes leading to falsified literals (note that we can also reuse the annotation from the first search), but also nodes leading to nodes contained in the first cut. If we can't find a cut entirely disjoint from the first one, we settle for an overlapping one, but we choose disjoint nodes as often as possible. Note that one necessarily finds a second cut if the first one was found: indeed in the worst case the two cuts are identical. Both algorithms were developed independently from [11] which was unavailable at the time, and private conversations with Jain indicate that his solutions are similar.

3.3 On-the-Fly Incremental CNF Translation

One of the contributions of Nenofar is that it combines non-clausal reasoning with conventional clausal reasoning in order to leverage the advantages of both approaches. Top-level conjuncts are considered separately and are handled by the clausal or the non-clausal reasoners depending on their nature. Additionally, Nenofar learns clauses during conflict analysis as most solvers do [16]. Finally, in our design, all clauses from the horizontal graphs which take part in implications

Algorithm 2 Search for an acceptable cut

```
1: function CutSearch(G_V(φ), A)                    ▷ A is the current assignment
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
```

are transfered to the clausal reasoner. This effectively acts as an incremental translation to CNF, since more and more conflicts and implications are detected outside the non-clausal reasoner as the search progresses. An indirect advantage of this choice is that the computation of implicants during conflict analysis can be done on clauses alone.

3.4 Combining Propositional and Theory Reasoning

To integrate propositional and theory reasoning, we leverage the existing DPLL scheme [3]. We observe that the original description of the interface of theory solvers does not require the use of a clausal boolean solver. We are not aware however of other work integrating such theory solvers with other boolean solvers. The communication between the theory solver and the boolean reasoners is not different from the original work on DPLL(T): the propositional solver communicates new truth value assignments to the theory solver, which either detects that the set of assignments is unsatisfiable, or returns a (possibly incomplete) set of consequences.

Algorithm 3 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, ∅, C_1)
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                          ▷ v must 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 ∈ S . leading(s) = ⊤ then
35:                leading(v) ← ⊤
36:            else
37:                marked(v) ← ⊤
38:            end if
39:        end if
40:    end foreach
41:    return (⊥, ∅)
42: end function
```

4 Implementation

Figure 7 shows a schema of the solving process in Nenofar. The preprocessing step converts the formula to NNF or to CNF if such a conversion is selected, and splits top level conjuncts.

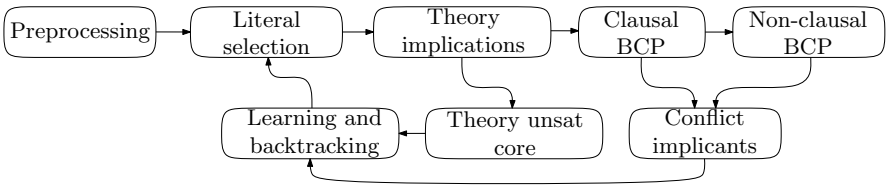

Fig. 7. Overview of the solving process

If as a result of this first phase there are no non-clausal constraints in the formula, Nenofar acts as a standard DPLL(T) solver. Otherwise, the NC(T) extension in Nenofar does boolean constraint propagation and conflict detection on the non-clausal constraints. Nenofar implements learning, and the learned formulas are clauses. Clauses are handled using the two-watched literal scheme as usual [15]. As already noted in the previous section, as the search progresses, the solver behaves more and more like a DPLL(T) solver.

Our implementation of the theory solver for the uninterpreted function symbols (QF_UF) is based on [3, 17]. Nenofar also implements other features commonly found in SMT solvers, such as the VSIDS literal selection heuristic [15], clause deletion, and restarts [18]. The system is implemented in C++ and the source code is freely available online at:

http://lara.epfl.ch/dokuwiki/nenofar

5 CNF Encodings

Solvers working on formulas in CNF need to include a translation phase to handle arbitrary formulas. Contrary to NNF translation which is straightforward, CNF translation needs to be done carefully; indeed, a naive application of De Morgan's laws and of the distributive properties of $\wedge$ and $\vee$ can lead to an exponential grow in the size of the formula. The typical way to work around this is due to Tseitin [7] and consists in introducing fresh boolean variables P_i to represent subformulas ϕ_i and then adding new clauses to encode the relationship $P_i \leftrightarrow \phi_i$. Applying this principle bottom-up, one can transform any boolean formula into an equisatisfiable set of clauses. This technique applied to (1) yields (2), which consists in 17 clauses and contains 5 fresh boolean variables. In the rest of this paper we will call this encoding $\text{CNF}_\top$.

$$
\begin{aligned}
&(P_0 \vee P_2) \wedge (\neg P_0 \vee P_1) \wedge (\neg P_0 \vee a = b) \wedge (\neg P_0 \vee a \neq c) \\
\wedge\ &(P_0 \vee \neg P_1 \vee a \neq b \vee a = c) \wedge (\neg P_1 \vee f(a) \neq f(b) \vee b = c) \\
\wedge\ &(P_1 \vee f(a) = f(b)) \wedge (P_1 \vee b \neq c) \wedge (\neg P_2 \vee b = f(b)) \wedge (\neg P_2 \vee P_3) \\
\wedge\ &(P_2 \vee b \neq f(b) \vee \neg P_3) \wedge (\neg P_3 \vee b = f(a) \vee P_4) \wedge (P_3 \vee \neg P_4) \\
\wedge\ &(P_3 \vee b \neq f(a)) \wedge (\neg P_4 \vee a = b) \wedge (\neg P_4 \vee b \neq f(a)) \\
\wedge\ &(P_4 \vee a \neq b \vee b = f(a))
\end{aligned}
\tag{2}
$$

Plaisted and Greenbaum [8] observed that if all subformulas are positive (as in NNF), a shorter encoding, which we will call CNF_{PG}, is possible, where the new constraints are relaxed to $P_i \rightarrow \phi_i$. While this reduces the number of new clauses compared to Tseitin's encoding, it doesn't have an effect on the number of variables. Formula (3) is (1) encoded using this techniques, and has 10 clauses.

$$
\begin{aligned}
&(P_0 \vee P_2) \wedge (\neg P_0 \vee P_1) \wedge (\neg P_0 \vee a = b) \wedge (\neg P_0 \vee a \neq c) \\
\wedge\ &(\neg P_1 \vee f(a) \neq f(b) \vee b = c) \wedge (\neg P_2 \vee b = f(b)) \wedge (\neg P_2 \vee P_3) \\
\wedge\ &(\neg P_3 \vee b = f(a) \vee P_4) \wedge (\neg P_4 \vee a = b) \wedge (\neg P_4 \vee b \neq f(a))
\end{aligned}
\tag{3}
$$

While *in general*, encoding a formula by distributing conjunctions over disjunctions and vice versa is worse than introducing a fresh variable, it is not *always* the case and sometimes the distribution leads to fewer clauses — and less variables, since none are introduced. In [9], Jackson and Sheridan give an algorithm which, for a given formula, determines which of its subformulas should be renamed using fresh boolean variables and which should be converted by distribution such that the resulting number of clauses is minimal. We will refer to this encoding as CNF_{JS}. Applying their procedure to (1) results in (4) which contains 6 clauses and only 1 boolean variable.

$$
\begin{aligned}
&(P_0 \vee b = c \vee f(a) \neq f(b)) \wedge (P_0 \vee a = b) \wedge (P_0 \vee a \neq c) \\
\wedge\ &(\neg P_0 \vee b = f(b)) \wedge (\neg P_0 \vee a = b \vee b = f(a)) \\
\wedge\ &(\neg P_0 \vee b \neq f(a) \vee b = f(a)
\end{aligned}
\tag{4}
$$

We should note that the impressive reduction observed on this particular example comes from its simple structure. In general, we observed a reduction in the number of clauses around 15% to 20% compared to the Plaisted-Greenbaum encoding, and an average reduction in the number of variables slightly above 50%.

6 Experimental Evaluation

We compared our NC(T) strategy against three different CNF encodings: CNF_{T}, CNF_{PG}, and CNF_{JS}.

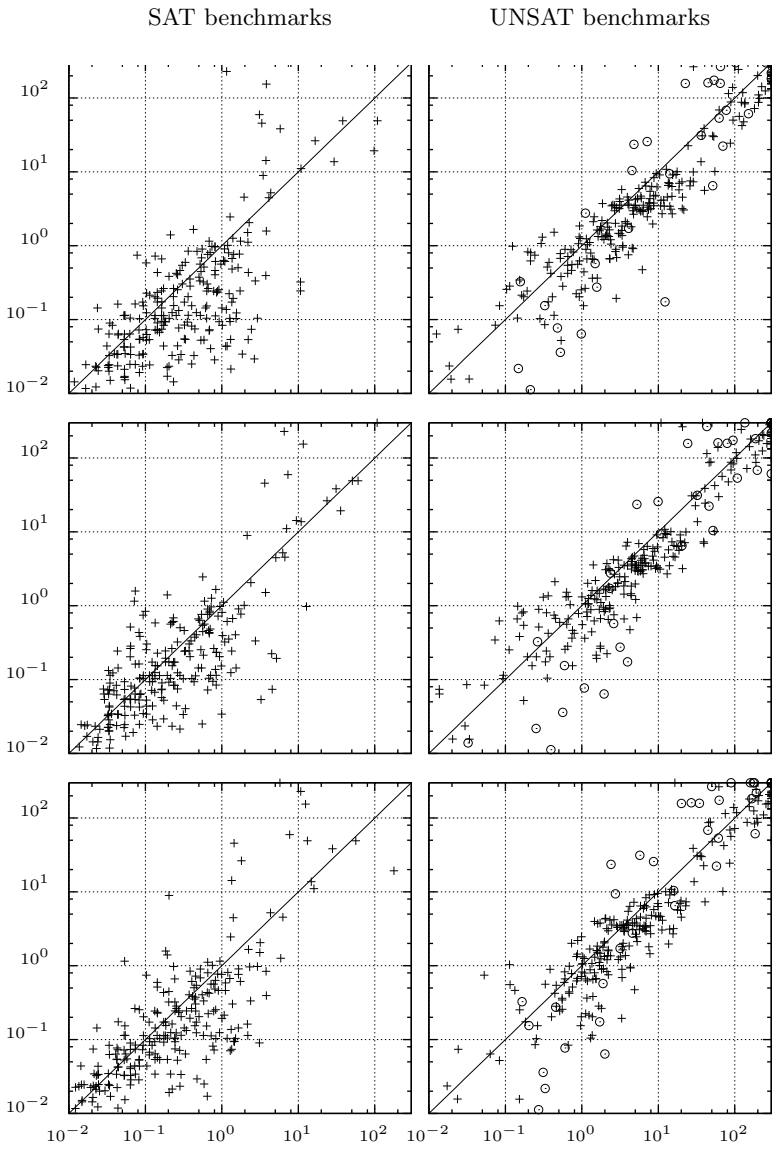

Fig. 8. Scatter plots of running times. The vertical axis is always the NC(T) strategy, the horizontal axes are, from top to bottom: $\mathrm{CNF}_{\mathsf{T}}$, $\mathrm{CNF}_{\mathsf{PG}}$, $\mathrm{CNF}_{\mathsf{JS}}$. The left column shows satisfiable benchmarks and unsatisfiable ones are on the right. Crosses are SMT-LIB benchmarks, circles are bounded model-checking benchmarks. Points below the diagonals indicate superior performance of NC(T).

We used two different benchmark sets. The SMT-LIB contains over 6000 non-clausal benchmarks for the theory QF_UF; we randomly picked 599 of them

		NC(T)		CNF_{T}		CNF_{PG}		CNF_{JS}	
		#best	#TO	#best	#TO	#best	#TO	#best	#TO
SMT-LIB	SAT (235)	99	1	38	0	52	0	46	0
	UNSAT (364)	129	127	35	146	39	143	36	143
	Bounded MC (41)	15	9	6	9	1	10	14	5
Overall (640)		243	137	79	155	92	153	96	148

Fig. 9. Experiments with 5 minute timeouts. "#best" is the count of times a strategy was faster than all others. "#TO" is the count of timeouts.

	NC(T)	CNF_{T}	CNF_{PG}	CNF_{JS}
NC(T)	–	162/195	145/202	150/192
CNF_{T}	73/78	–	105/116	98/103
CNF_{PG}	90/70	130/139	–	124/120
CNF_{JS}	85/82	137/156	111/139	–

Fig. 10. Comparison of all pairs of strategies. An entry (i, j) indicates how many times the strategy i outperformed j, on SAT and UNSAT problems. For instance, NC(T) was faster than CNF_{JS} on 150 SAT and 192 UNSAT benchmarks, for a total of 342.

to constitute our first set. The second set consists of verification conditions generated from bounded model-checking (using loop unrolling) of programs manipulating linked data structures. All benchmarks in that second set were obtained from valid verification conditions and are thus unsatisfiable.

We ran the tests on a 2 quad-core Xeon 2.66 GHz/16GB RAM running Debian/Linux 64bit. Nenofar uses no parallelization and we observed similar results on standard desktop machines.

Figure 8 shows the running times for NC(T) compared to the other strategies, with satisfiable and unsatisfiable benchmarks displayed separately, and Figure 9 summarizes these results. Overall, the NC(T) strategy resulted in the fewest timeouts and was the fastest one on more benchmarks than any other strategy. Figure 10 displays a pairwise comparison of the tested strategies.

Some benchmarks from the bounded model-checking set turned out to be harder to solve for NC(T) than for any CNF encoding. It should be noted that these benchmarks contain a large proportion of clauses in their original form. We believe that in these cases, most of the implications and conflicts are derived from the clauses and that the extra time spent repeatedly analyzing the few non-clausal constraints is detrimental to the NC(T) strategy. We did not observe similar results on benchmarks constituted of only or mostly non-clausal constraints.

7 Related Work

To the best of our knowledge, this is the first non-clausal SMT solver. We present relevant work from the SAT community. Most of the research on non-clausal reasoning can be classified between work on non-clausal solvers and work on efficient CNF encodings.

Earlier work by Jain et al. [14] introduced another graph-based non-clausal SAT solver. Rather than using DPLL-like recursive search, the approach consists in searching directly for a consistent 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 subsequently constrain the paths going through them. [11], which is the boolean basis for our integration and which we presented in the previous sections, in an extension of [14].

Thiffault et al. present another non-clausal SAT solver in [10]. The approach doesn't require converting to NNF and works by adapting the two-watched literal principle 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.

An earlier presentation of a generalization of the DPLL algorithm to non-clausal formulas is given in [19]. 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 adapted 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, which is essentially the principle underlying the $DPLL(\mathrm{T})$ scheme, although no mention of communication from the theory decision procedure to the boolean solver is made. As far as we know, their technique has only been evaluated on random problems.

To the best of our knowledge, no SMT solver performs boolean reasoning on non-clausal constraints. Z3 does make use of non-clausal information to reduce the amount of work for the theory solvers [20].

We presented part of [9] in Section 5. This work by Jackson and Sheridan also contains efficient constructions for equivalences ($\leftrightarrow$) and if-then-else subformulas. We did not present or implement these for practical reasons: none of our benchmarks contained any of these constructs.

Manolios and Vroon [21] present a similar, improved, encoding, although it appears that the main benefits apply only to equivalences and if-then-elses, the construction being essentially identical to [9] in their absence.

Finally, [22] presents preprocessing techniques for SAT which aim at simplifying a set of clauses. While preprocessing may seem to be a viable alternative to sophisticated CNF encodings, it has been shown that both approaches are complementary, rather than alternatives [23]. Nenofar does not implement preprocessing and we are not aware of preprocessing work targeted specifically at SMT solvers.

8 Conclusions

We presented the combination of non-clausal techniques with theory reasoning and with standard clausal reasoning, thus constituting the first non-clausal SMT solver. We implemented our system and compared it with several CNF encodings. Our observation is that reasoning on non-clausal constraints and applying an incremental translation works faster on most examples. Nenofar and the benchmarks used are available for further study.

References

1. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2008) 337–340
2. Nieuwenhuis, R., Oliveras, A.: Decision Procedures for SAT, SAT Modulo Theories and Beyond. The BarcelogicTools. (Invited Paper). In Sutcliffe, G., Voronkov, A., eds.: 12h International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'05. Volume 3835 of Lecture Notes in Computer Science., Springer (2005) 23–46
3. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast Decision Procedures. Proceedings of the 16th International Conference on Computer Aided Verification, CAV **4** (2004) 175–188
4. Barrett, C., Ranise, S., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). http://www.SMT-LIB.org (2009)
5. Flanagan, C., Saxe, J.B.: Avoiding Exponential Explosion: Generating Compact Verification Conditions. In: Proc. 28th ACM POPL. (2001)
6. Barnett, M., Leino, K.R.M.: Weakest-Precondition of Unstructured Programs. In: PASTE. (2005) 82–87
7. Tseitin, G.: On the Complexity of Derivation in Propositional Calculus. Studies in Constructive Mathematics and Mathematical Logic **2** (1968) 115–125
8. Plaisted, D., Greenbaum, S.: A Structure-preserving Clause Form translation. Journal of Symbolic Computation **2**(3) (1986) 293–304
9. Jackson, P., Sheridan, D.: Clause Form Conversions for Boolean Circuits. Lecture Notes in Computer Science **3542** (2005) 183–198
10. Thiffault, C., Bacchus, F., Walsh, T.: Solving Non-Clausal Formulas with DPLL Search. CP (2004)
11. Jain, H.: Verification Using Satisfiability Checking, Predicate Abstraction, and Craig Interpolation. PhD thesis, Carnegie Mellon University, School of Computer Science (2008)
12. Suter, P.: Non-Clausal Satisfiability Modulo Theories. Master's thesis, EPFL (2008) http://infoscience.epfl.ch/record/126445.
13. Andrews, P.B.: An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof. 2nd edn. Springer (Kluwer) (2002)
14. Jain, H., Bartzis, C., Clarke, E.: Satisfiability Checking of Non-clausal Formulas using General Matings. 9th International Conference on Theory and Applications of Satisfiability Testing (2006)
15. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. Proceedings of the 38th conference on Design automation (2001) 530–535

16. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. International Conference on Computer-Aided Design (ICCAD01) (2001) 279–285
17. Nieuwenhuis, R., Oliveras, A.: Proof-Producing Congruence Closure. Term Rewriting and Applications: 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005: Proceedings (2005)
18. Eén, N., Sörensson, N.: An Extensible SAT-solver. SAT **2919** (2003) 502–518
19. Armando, A., Giunchiglia, E.: Embedding Complex Decision Procedures Inside an Interactive Theorem Prover. Annals of Mathematics and Artificial Intelligence **8**(3) (1993) 475–502
20. de Moura, L., Bjørner, N.: Relevancy Propagation. Technical Report MSR-TR-2007-140, Microsoft Research (2007)
21. Manolios, P., Vroon, D.: Efficient Circuit to CNF Conversion. Lecture Notes in Computer Science **4501** (2007) 4–9
22. Eén, N., Biere, A.: Effective Preprocessing in SAT Through Variable and Clause Elimination. SAT (2005) 61–75
23. Eén, N., Mishchenko, A., Sörensson, N.: Applying Logic Synthesis for Speeding Up SAT. Lecture Notes in Computer Science **4501** (2007) 272–285