ECAI 2010
H. Coelho et al. (Eds.)
IOS Press, 2010

doi:10.3233/978-1-60750-606-5-21

Extending Clause Learning DPLL with Parity Reasoning

Tero Laitinen [1] and **Tommi Junttila** [1] and **Ilkka Niemelä** [1]

Abstract. We consider a combined satisfiability problem where an instance is given in two parts: a set of traditional clauses extended with a set of parity (xor) constraints. To solve such problems without translation to CNF, we develop a parity constraint reasoning method that can be integrated to a clause learning solver. The idea is to devise a module that offers a decision procedure and implied literal detection for parity constraints and also provides clausal explanations for implied literals and conflicts. We have implemented the method and integrated it to a state-of-the-art clause learning solver. The resulting system is experimentally evaluated and compared to state-of-the-art solvers.

1 Introduction

Solver technology for propositional satisfiability (SAT) has developed rapidly over the last decade and in many application areas such as hardware model checking, planning, software verification, combinatorial designs, and automated test pattern generation SAT solvers provide a state-of-the-art solution technique [4]. In this approach a problem instance is solved by encoding it as a propositional formula such that the models of the formula correspond to the solutions of the problem instance. Then a SAT solver can be used as an efficient search engine to find solutions to the problem instance. The most efficient SAT solvers require their input to be in conjunctive normal form (CNF). However, in many application areas such as circuit verification, bounded model checking, and logical cryptanalysis a substantial part of the encoding are parity (xor) constraints [3].

In the basic SAT approach such xor-clauses are translated to CNF, i.e., as traditional or-clauses, for the solver but this leads to a less compact encoding where the structure of xor-clauses is lost. For example, if the encoding consists of xor-clauses only, such an instance can be solved in polynomial time using Gaussian elimination. However, if such an instance is translated to CNF, even state-of-the-art clausal SAT solvers can scale very poorly [9].

In this paper the goal is to develop an extended SAT solver that can work with two kinds of clauses: or-clauses and xor-clauses. The aim is to take advantage of the state-of-the-art clausal solver technology based on conflict-driven clause learning [13] and to be able to exploit the structure and special properties of xor-clauses. We employ a framework similar to the DPLL(T) approach to Satisfiability Modulo Theories (SMT) (see e.g. [6, 1, 8, 14, 2]), where xor-clauses are handled by a *xor-reasoning module*. As the formal basis for the module we devise a powerful yet efficiently implementable proof system that captures unit propagation and equivalence reasoning for xor-clauses and introduce a method for computing explanations for literals derivable in the proof system. Using these results we develop a xor-reasoning module which offers implied literal detection and clausal explanations for enabling conflict-driven clause learning techniques to be extended to xor-clauses. For integrating the module into state-of-the-art clausal SAT solver technology we devise a variant of the DPLL(T) framework that also handles the fact that in our setting the or-clauses and the xor-clauses (the "theory part") can have shared variables.

Related work. There is a considerable amount of previous work on extending clausal solvers with xor (equivalence) reasoning techniques. Baumgartner and Massacci [3] develop a decision method for SAT problems where or-clauses and xor-clauses can be combined. Their work is based on the standard DPLL procedure without conflict-driven learning. EqSatz [12] recognizes binary and ternary equivalences in a CNF formula and performs substitutions using a set of inference rules. The equivalence reasoning is tightly integrated in the solver and is performed after unit propagation. The solver march_eq [10] extracts equivalences from a CNF formula and uses them in pre-processing as well as during the search. However, neither of these approaches support conflict-driven clause learning. The solver MoRsat [5] extracts equivalences from CNF, too. Such constraints are stored as normal clauses and efficient unit propagation is supported with special watched literal techniques. CryptoMiniSat [16] accepts a mixture of or-clauses and xor-clauses as its input. It has special data structures for xor-clauses and performs Gaussian elimination after a specified number of literals have been assigned and no other propagation rules can be fired. The solver lsat [15] performs preprocessing that reconstructs structural information (including equivalences) from the CNF formula which is exploited during the search. The inference rules of our proof system are similar to those in [3] and also include a substitution rule for binary xor-clauses as in EqSatz [12]. The main difference in our approach is the combination of equivalence reasoning and conflict-driven clause learning through the use of "lazy" DPLL(T) style integration of the xor-reasoning module to a SAT solver.

2 Preliminaries

We first define some basic notation needed in the rest of the paper.

An *atom* is either a propositional variable or the special symbol $\top$ which denotes the constant "true". A *literal* is an atom A or its negation $\neg A$; we identify $\neg\top$ with $\bot$ and $\neg\neg A$ with A. A traditional, non-exclusive *or-clause* is a disjunction $l_1 \vee \cdots \vee l_n$ of literals. A *xor-clause* is an expression of form $l_1 \oplus \cdots \oplus l_n$, where $l_1, \ldots, l_n$ are literals and the symbol $\oplus$ stands for the exclusive logical or. A *clause* is either an or-clause or a xor-clause.

A *truth assignment* π is a set of literals such that $\top \in \pi$ and $\forall l \in \pi : \neg l \notin \pi$. We define the "satisfies" relation $\models$ between

[1] Aalto University, Department of Information and Computer Science, P.O. Box 15400, FI-00076 Aalto, Finland. email: {Tero.Laitinen,Tommi.Junttila,Ilkka.Niemela}@tkk.fi. The financial support of the Academy of Finland (project 122399) is gratefully acknowledged.

a truth assignment π and logical constructs as follows: (i) if l is a literal, then $\pi \models l$ iff $l \in \pi$, (ii) if $C = (l_1 \vee \cdots \vee l_n)$ is an or-clause, then $\pi \models C$ iff $\pi \models l_i$ for some $l_i \in \{l_1, \ldots, l_n\}$, and (iii) if $C = (l_1 \oplus \cdots \oplus l_n)$ is a xor-clause, then $\pi \models C$ iff π is total for C (i.e. $\forall 1 \leq i \leq n : l_i \in \pi \vee \neg l_i \in \pi$) and $\pi \models l_i$ for an odd number of literals of C. Note that literal duplication in xor-clauses makes a difference, e.g. $\{x, \neg y\}$ satisfies $x \oplus y$ but not $x \oplus y \oplus x$. However, the order of the literals is insignificant and thus, e.g., $(x \oplus y \oplus x)$ and $(x \oplus x \oplus y)$ are considered to be the same xor-clause. Furthermore, observe that no truth assignment satisfies the empty or-clause $()$ or the empty xor-clause $()$, i.e. these clauses are synonyms for $\bot$.

A *cnf-xor formula* ϕ is a conjunction of clauses, expressible as a conjunction

$$\phi = \phi_{\text{or}} \wedge \phi_{\text{xor}}, \tag{1}$$

where ϕ_{or} is a conjunction of or-clauses and ϕ_{xor} is a conjunction of xor-clauses. A truth assignment π *satisfies* ϕ, denoted by $\pi \models \phi$, if it satisfies each clause in it; ϕ is called *satisfiable* if there exists such a truth assignment satisfying it, and *unsatisfiable* otherwise. The *cnf-xor satisfiability* problem studied in this paper is to decide whether a given cnf-xor formula has a satisfying truth assignment. As usual, a set of clauses $\{C_1, ..., C_n\}$ is interpreted as the formula $\bigwedge_{i=1}^{n} C_i$. A formula ϕ' is a *logical consequence* of a formula ϕ, denoted by $\phi \models \phi'$, if $\pi \models \phi$ implies $\pi \models \phi'$ for all truth assignments π. Let A be an atom different from $\top$ and C, D be xor-clauses. We use $C\,[A/D]$ to denote the xor-clause that is identical to C except that all occurrences of A in C are substituted with D once. For instance, $(x_1 \oplus x_1 \oplus x_2)\,[x_1/(x_1 \oplus x_3)] = x_1 \oplus x_3 \oplus x_1 \oplus x_3 \oplus x_2$.

Normal form for xor-clauses. In the rest of the paper, we implicitly assume that each xor-clause is given in a *normal form* such that (i) each atom occurs at most once in it, and (ii) all the literals in it are positive i.e. the negation does not appear. Especially, a xor-clause resulting from a substitution $C\,[A/X]$ is implicitly transformed to the normal form. Given any xor-clause, its unique normal form can be obtained by applying the following rewrite rules (where C is a possibly empty xor-clause and A is an atom) in any order until saturation is reached: (i) $\neg A \oplus C \rightsquigarrow A \oplus \top \oplus C$, and (ii) $A \oplus A \oplus C \rightsquigarrow C$. These rules are similar to those given in [3]. For instance, the normal form of $\neg x_1 \oplus x_2 \oplus x_3 \oplus x_3$ is $x_1 \oplus x_2 \oplus \top$, while the normal form of $x_1 \oplus x_1$ is the empty xor-clause $()$. We say that a xor-clause is *unary* if it is either of form x or $x \oplus \top$ for some variable x; we will identify $x \oplus \top$ with the literal $\neg x$.

3 DPLL(XOR)

Recall that our goal is to solve cnf-xor formulas of form $\phi = \phi_{\text{or}} \wedge \phi_{\text{xor}}$. In this paper we take an approach similar to the so-called "lazy", or DPLL(T), approach to Satisfiability Modulo Theories (SMT) (see e.g. [6, 1, 8, 14, 2]). In a nutshell, the basic idea is that we use a state-of-the-art, conflict-driven and clause learning (CDCL) SAT solver (see e.g. [13]) for finding satisfying truth assignments for the or-part ϕ_{or} and interface it with a *xor-reasoning module* that validates whether the (partial or total) truth assignment under consideration is consistent with the constraints in the xor-part ϕ_{xor}. The simplified schema for the approach is shown in Fig. 1; the standard SAT solver part (lines 2–4 and 13–19) iteratively builds a truth assignment π by selecting an unassigned variable, unit propagating its effects and performing conflict analysis when an inconsistency is found. The xor-module is interfaced at lines 1 and 6–12; its methods and functionalities discussed below are similar to those provided by "theory solvers" in the DPLL(T) approach for SMT:

```
1.  initialize xor-module M with φ_xor
2.  π = ⟨⟩        /*the truth assignment*/
3.  while true:
4.      (π', confl) = UNITPROP(φ_or, π)    /*standard unit propagation*/
5.      if not confl:        /*apply xor-reasoning*/
6.          for each literal l in π' but not in π: M.ASSIGN(l)
7.          (l̂_1, ..., l̂_k) = M.DEDUCE()
8.          for i = 1 to k:
9.              let C = M.EXPLAIN(l̂_i)
10.             if l̂_i = () or ¬l̂_i ∈ π': confl = C, break
11.             else if l̂_i ∉ π: add l^C to π'
12.         if k > 0 and not confl: continue    /*unit propagate further*/
13.     let π = π'
14.     if confl:        /*standard Boolean conflict analysis*/
15.         analyze conflict, learn a conflict clause
16.         backjump or return UNSAT if not possible
17.     else:
18.         add a heuristically selected unassigned literal in φ_or to π
19.         or return SAT if no such variable exists
```

Figure 1. The essential skeleton of DPLL(XOR)

1. When initialized (at line 1 in Fig. 1), the module receives the xor-part ϕ_{xor} of the formula ϕ. The clauses in ϕ_{xor} are from now on called the *original xor-clauses*.
2. The SAT solver communicates the (possibly partial) truth assignment it has built for ϕ_{or} to the module by calling its ASSIGN method for each literal in the assignment (line 6). These literals are called the *xor-assumptions*.
3. The core functionality of the module is the deduction and consistency checking method DEDUCE (at line 7). If the xor-assumptions (i.e. unary xor-clauses) $l_1, ..., l_k$ have been communicated to the module earlier, the method performs deduction on the augmented xor-formula
$$\phi_{\text{xor}} \wedge l_1 \wedge ... \wedge l_k.$$
As the result, the method returns a list $l'_1, ..., l'_m$ of *xor-implied literals* that are logical consequences of $\phi_{\text{xor}} \wedge l_1 \wedge ... \wedge l_k$. Thus, the SAT solver can extend its current partial truth assignment with these literals (at line 11). Observe especially that if a xor-implied literal l_i is the false literal $\bot$, i.e. the empty clause $()$, then $\phi_{\text{xor}} \wedge l_1 \wedge ... \wedge l_k$ is unsatisfiable and thus the current partial truth assignment built in the SAT solver cannot be extended to an assignment satisfying $\phi_{\text{or}} \wedge \phi_{\text{xor}}$.
4. Upon deriving a conflict, the SAT solver analyzes the partial truth assignment π to find out which literals in it are responsible for the conflict (at lines 14–15). For this purpose π is actually ordered and the literals in it (except for the decision literals added in line 18) are annotated literals of form l^C, where C is an or-clause of form $(l_1 \vee ... \vee l_k \vee l)$ such that (i) C is a logical consequence of $\phi_{\text{or}} \wedge \phi_{\text{xor}}$, and (ii) l is implied by C in π, meaning that the literals $\neg l_1, ..., \neg l_k$ must appear before l in π. To build such an annotation for a xor-implied literal l returned by DEDUCE, the EXPLAIN method (called at line 9) returns an or-clause $C = (l_1 \vee ... \vee l_k \vee l)$ such that (i) C is a logical consequence of ϕ_{xor}, and (ii) the literals $\neg l_1, ..., \neg l_k$ have either been assigned to the xor-module (and are therefore in π') or have been returned as xor-implied literals before (and are thus in π' when l is inserted at line 11). Such or-clauses are called *xor-explanations*.
Observe that the conflict analysis at line 15 builds a new or-clause

$$\oplus\text{-Unit}^{+}\colon \frac{A \qquad C}{C\,[A/\top]} \qquad \oplus\text{-Unit}^{-}\colon \frac{A \oplus \top \qquad C}{C\,[A/\bot]}$$

$$\oplus\text{-Eqv}^{+}\colon \frac{A_1 \oplus A_2 \oplus \top \qquad C}{C\,[A_1/A_2]} \qquad \oplus\text{-Eqv}^{-}\colon \frac{A_1 \oplus A_2 \qquad C}{C\,[A_1/(A_2 \oplus \top)]}$$

Figure 2. Inference rules of the proof system

that is a logical consequence of $\phi_{\text{or}} \wedge \phi_{\text{xor}}$. This clause is usually *learnt*, i.e. augmented to the formula ϕ_{or}, as it is built in a way that prevents similar conflicts from arising again. As the xor-explanations found at line 9 are logical consequences of ϕ as well, they could also be learnt in the same way. We will return to this issue in Sect. 3.3.

5. For the purpose of backtracking, the xor-module also implements the following methods: (i) ADD-BACKJUMP-POINT, which records the state of the module (including the sequence of xor-assumptions given so far) and returns an associated backjump point, and (ii) BACKJUMP, which restores the state associated with a previously added backjump point (this includes discarding the xor-assumptions communicated after setting the backjump point).

In the rest of the section, we first describe the proof system used as the deduction engine in the xor-module, and then how explanations are computed for xor-implied literals. We also introduce and analyze different ways to handle the case when not all variables in the xor-part appear in the or-part handled by the CDCL SAT solver.

3.1 The Proof System

When the DEDUCE method of the xor-module is invoked, it applies the following *proof system* to the conjunction $\psi = \phi_{\text{xor}} \wedge l_1 \wedge ... \wedge l_k$ of the original xor-clauses ϕ_{xor} and the xor-assumptions $l_1, ..., l_k$. The proof system produces new xor-clauses that are logical consequences of ψ. Thus, the unary xor-clauses produced are xor-implied literals and the deduction of the empty clause signals that ψ is unsatisfiable.

The *inference rules* of the proof system are listed in Fig. 2, where A, A_1 and A_2 are atoms different from $\top$ and C is a xor-clause. For instance, if we have the xor-clauses $x_1 \oplus x_2$ and $x_1 \oplus x_2 \oplus x_3 \oplus \top$, then the $\oplus$-Eqv$^-$ rule allows us to derive $x_2 \oplus \top \oplus x_2 \oplus x_3 \oplus \top$, i.e., x_3 in the normal form. First, notice that the rules are sound in the following sense: if $R\colon \frac{X \qquad Y}{Z}$ is an instance of one of the rules, then the consequent Z is a logical consequence of the premises X and Y, i.e. $\{X, Y\} \models Z$. Observe that these rules are basically the "$\oplus$-Unit" and "$\oplus$-Eqv" rules appearing in [3]. Thus, our proof system is a subset of the one in [3]; in particular, we do not apply the Gaussian elimination rule of [3] as our goal is to have a lightweight system that is easier to implement efficiently.

Let ψ be any set of xor-clauses, e.g. the conjunction $\phi_{\text{xor}} \wedge l_1 \wedge ... \wedge l_k$ of original xor-clauses and xor-assumptions discussed earlier. A *xor-derivation* on ψ is a vertex-labeled directed acyclic graph $G = \langle V, E, L\rangle$ such that the vertex set V is finite, the labeling function L assigns each vertex $v \in V$ a xor-clause $L(v)$ and the following hold for each vertex $v \in V$:

1. v has no incoming edges (i.e. v is an *input vertex*) and is labeled with a xor-clause $L(v) \in \psi$, or
2. v has exactly two incoming edges, originating from some vertices v_1 and v_2, and the xor-clause $L(v)$ has been derived from $L(v_1)$ and $L(v_2)$ by using one of the inference rules. That is, $R\colon \frac{L(v_1) \qquad L(v_2)}{L(v)}$ must be an instance of an inference rule R appearing in Fig. 2.

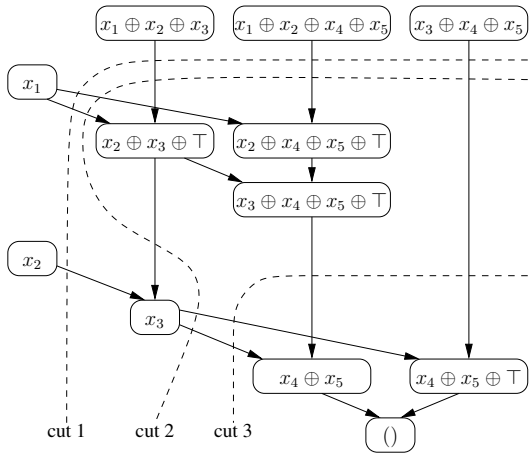

Figure 3. A xor-refutation.

We say that a xor-clause C is *derivable* from ψ in the proof system, denoted by $\psi \vdash C$, if there exists a xor-derivation on ψ that contains a vertex labeled with C. As a direct consequence of the definition of xor-derivations and the soundness of the inference rules, it holds that if a xor-derivation on ψ contains a vertex labeled with the xor-clause C, then C is a logical consequence of ψ.

Lemma 1. $\psi \vdash C$ implies $\psi \models C$.

A xor-derivation on ψ is a *xor-refutation of* ψ if it contains a vertex labeled with the empty clause $()$; in this case, ψ is unsatisfiable.

Example 1. Figure 3 shows a xor-refutation of $\psi = \{(x_1 \oplus x_2 \oplus x_3), (x_1 \oplus x_2 \oplus x_4 \oplus x_5), (x_3 \oplus x_4 \oplus x_5), x_1, x_2\}$. (Ignore the dotted lines for now). In the figure, the labeling clause of each vertex is drawn inside it. Recall that all clauses are implicitly transformed into the normal form; for instance, at the last step we apply the $\oplus$-Eqv$^-$ rule to derive $(x_4 \oplus x_5 \oplus \top)\,[x_4/x_5 \oplus \top] = x_5 \oplus \top \oplus x_5 \oplus \top = ()$.

Note that the proof system is not refutationally complete, meaning that there are unsatisfiable clause sets for which there are no xor-refutations in the proof system. As a simple example, consider the clause set $\{(x_1 \oplus x_2 \oplus x_3), (x_1 \oplus x_2 \oplus x_3 \oplus \top)\}$ which is unsatisfiable but the largest xor-derivation on it consists only of two vertices labeled with the clauses in the set. Thus, the proof system is not complete, meaning that $\psi \models C$ does not necessarily imply $\psi \vdash C$. However, the proof system is *eventually refutationally complete* in the following sense: if the set ψ of xor-clauses contains a unary clause (x) or $(x \oplus \top)$ for each variable x occurring in ψ, then the empty clause is derivable if and only if ψ is unsatisfiable. In the context of our DPLL(XOR) approach this means that if the SAT solver has provided a xor-assumption l_i for each variable occurring in ϕ_{xor}, then the xor-module is indeed able to decide whether $\phi_{\text{xor}} \wedge l_1 \wedge ... \wedge l_n$ is satisfiable or not.

If we compare the deduction capabilities of our proof system on a set of xor-clauses to the those of the standard unit propagation on the corresponding or-clause formula, we observe that (i) the

$\oplus$-Unit$^+$ and $\oplus$-Unit$^-$ rules simulate unit propagation while (ii) the $\oplus$-Eqv$^+$ and $\oplus$-Eqv$^-$ rules allow the proof system to deduce consequences that the unit propagation cannot. As an example, given the set $\{(x \oplus y \oplus \top), (x \oplus a \oplus b), (y \oplus b \oplus c), (a)\}$ of xor-clauses, the proof system can derive the unit clause (c) but the standard unit propagation rule cannot deduce (c) on the corresponding or-clause formula $(\neg x \vee y) \wedge (x \vee \neg y) \wedge (x \vee a \vee b) \wedge (\neg x \vee \neg a \vee b) \wedge (\neg x \vee a \vee \neg b) \wedge (x \vee \neg a \vee \neg b) \wedge (y \vee b \vee c) \wedge (\neg y \vee \neg b \vee c) \wedge (\neg y \vee b \vee \neg c) \wedge (y \vee \neg b \vee \neg c) \wedge (a)$.

Incrementality. Recall that in our DPLL(XOR) approach the xor-module has a constant set of original xor-clauses ϕ_{xor} and a dynamically increasing or decreasing sequence of xor-assumptions $l_1 \wedge ... \wedge l_k$. Now observe that xor-derivations are monotonic in the following sense: if $G = \langle V, E, L\rangle$ is a xor-derivation on ψ, then $G = \langle V, E, L\rangle$ is a xor-derivation on $\psi \wedge C$ for any xor-clause C. Therefore, if the xor-module has already built a xor-derivation for $\phi_{\text{xor}} \wedge l_1 \wedge ... \wedge l_k$ and receives a new xor-assumption l_{k+1} from the SAT solver, it can extend the derivation by adding a vertex labeled with l_{k+1} and continue applying appropriate inference rules. Similarly, when xor-assumptions are retracted with the BACKJUMP method, the module can simply remove the vertices introduced since the corresponding ADD-BACKJUMP-POINT method call.

3.2 Computing Explanations

We now describe how explanations for xor-implied literals are computed from xor-derivations.

Assume a fixed xor-derivation $G = \langle V, E, L\rangle$ on a set ψ of xor-clauses. As usual, we say that a vertex $v' \in V$ is a descendant of a vertex $v \in V$ if $\langle v, v'\rangle$ is in the transitive closure of the edge relation E; in such a case, v is an ancestor of v'. A *cut* of G is a partitioning $W = \langle V_{\text{a}}, V_{\text{b}}\rangle$ of V, i.e. $V = V_{\text{a}} \cup V_{\text{b}}$ and $V_{\text{a}} \cap V_{\text{b}} = \emptyset$, such that all the input vertices of G belong to V_{a}. The set V_{a} is called the *premise part* of the cut while V_{b} is the *consequent part*. Given a non-input vertex $v \in V$, a *cut* for v is a cut $\langle V_{\text{a}}, V_{\text{b}}\rangle$ of G such that $v \in V_{\text{b}}$. If $W = \langle V_{\text{a}}, V_{\text{b}}\rangle$ is a cut of G, then the *reason set* of the cut is the set of those vertices in V_{a} that have an edge to V_{b}, i.e. $\text{reason}(W) = \{v \in V_{\text{a}} \mid \exists v' \in V_{\text{b}} : \langle v, v'\rangle \in E\}$. Now the conjunction of the xor-clauses in the reason set vertices imply the ones labeling the vertices in V_{b}:

Lemma 2. If $W = \langle V_{\text{a}}, V_{\text{b}}\rangle$ is a cut for a non-input vertex v, then $(\bigwedge_{v' \in \text{reason}(W)} L(v')) \models L(v)$.

Proof. Consider the set of vertices $V' = \text{reason}(W) \cup \{t \in V \mid t \text{ is a descendant of a vertex in } \text{reason}(W)\}$. Now $G' = \langle V', E \cap V' \times V', L|_{V'}\rangle$, where $L|_{V'}$ is the restriction of L to V', is a xor-derivation on $\{L(u) \mid u \in \text{reason}(W)\}$ and includes the vertex v. Thus, $\{L(u) \mid u \in \text{reason}(W)\} \vdash L(v)$ and, by Lemma 1, $\{L(u) \mid u \in \text{reason}(W)\} \models L(v)$. □

Example 2. Consider again the xor-derivation in Fig. 3 and the cut "cut 2". The corresponding reason set consists of the vertices labeled with the xor-clauses in $\psi = \{(x_1 \oplus x_2 \oplus x_3), (x_1 \oplus x_2 \oplus x_4 \oplus x_5), (x_3 \oplus x_4 \oplus x_5), x_3, x_1\}$. As the vertex labeled with the empty clause is in the consequent part of the cut, $\psi \models ()$ i.e. ψ is unsatisfiable.

CNF-compatible cuts. In the context of our DPLL(XOR) approach, of particular interest are the cuts where the reason set consists only of vertices labeled with original or unary xor-clauses. That is, assume that $G = \langle V, E, L\rangle$ is a xor-derivation on $\phi_{\text{xor}} \wedge l_1 \wedge ... \wedge l_k$, where ϕ_{xor} is the set of original xor-clauses and l_i are xor-assumptions. In this setting, we say that a cut $W = \langle V_{\text{a}}, V_{\text{b}}\rangle$ is *cnf-compatible* if for each vertex w in the corresponding reason set $\text{reason}(W)$ it holds that either

1. w is an input vertex (in which case $L(w)$ is an original xor-clause in ϕ_{xor} or an unary xor-assumption clause), or
2. $L(w)$ is a unary xor-clause.

Now suppose that a vertex v labeled with a unary xor-clause $L(v)$ belongs the consequent part of a cnf-compatible cut $W = \langle V_{\text{a}}, V_{\text{b}}\rangle$. Let $A = \{L(v') \mid v' \in \text{reason}(W) \wedge L(v') \in \phi_{\text{xor}}\}$ and $B = \{L(v') \mid v' \in \text{reason}(W) \wedge L(v') \notin \phi_{\text{xor}}\}$ be the set of xor-clauses labeling the reason set vertices partitioned into those occurring in ϕ_{xor} (the set A) and the others (the set B). As W is cnf-compatible, the set $B = \{b_1, ..., b_m\}$ consists only of unary clauses. By Lemma 2 we have that $A \wedge B \models L(v)$, implying that $A \models (\neg b_1 \vee ... \vee \neg b_m \vee L(v))$ and, by the fact $A \subseteq \phi_{\text{xor}}$, that $\phi_{\text{xor}} \models (\neg b_1 \vee ... \vee \neg b_m \vee L(v))$. Therefore, *cnf-compatible cuts for vertices labeled with unary xor-clauses allow us to derive or-clauses that capture parts of derivations allowed in our proof system*. Observe that for any non-input vertex v, there is a unique cnf-compatible cut $W = \langle V_{\text{a}}, V_{\text{b}}\rangle$ for v that is minimal with respect to the size of the consequent part V_{b}; this is the cut where V_{b} is the smallest set U satisfying (i) $v \in U$ and (ii) if $v' \in U$, $v'' \in V$, $\langle v'', v'\rangle \in E$, v'' is not an input vertex, and $L(v'')$ is not unary, then $v'' \in U$. We call such a cut the *closest cnf-compatible cut* for v.

Example 3. The cuts "cut 1" and "cut 2" in Fig. 3 are cnf-compatible but "cut 3" is not. The cut "cut 2" is the closest cnf-compatible cut for the vertex labeled with the empty clause ().

3.3 Handling XOR-internal variables

Assume an instance $\phi = \phi_{\text{or}} \wedge \phi_{\text{xor}}$ having variables that occur in the xor-part ϕ_{xor} but not in the or-part ϕ_{or} (we call such variables *xor-internal* as opposed to *xor-shared* variables common to ϕ_{or} and ϕ_{xor}). Now if the SAT solver only sees the variables in the or-part ϕ_{or}, it may happen that it constructs a truth assignment that satisfies ϕ_{or}, communicates all the literals $l_1, ..., l_n$ in the assignment to the xor-module as xor-assumptions, and asks the xor-module (by calling the DEDUCE method) whether $\phi_{\text{xor}} \wedge l_1 \wedge ... \wedge l_n$ is satisfiable. But because the proof system of the xor-module is not refutationally complete and there are xor-internal variables not assigned by the xor-assumptions, it is possible that the xor-module cannot deduce whether $\phi_{\text{xor}} \wedge l_1 \wedge ... \wedge l_n$ is satisfiable. We consider three approaches to solve this problem:

1. Implement a DEDUCE-FULL method that performs Gaussian elimination on $\phi_{\text{xor}} \wedge l_1 \wedge ... \wedge l_n$, thus solving its satisfiability. As Gaussian elimination seems to be more difficult to implement efficiently than our proof system, DEDUCE-FULL should preferably be called only when there are no unassigned xor-shared variables. A major drawback of this approach is, as the SAT solver does not see xor-internal variables, that the xor-internal xor-implied literals returned by the EXPLAIN method are of no use. Furthermore, the clauses returned by EXPLAIN can contain only xor-shared variables; thus we cannot use the closest cnf-compatible cuts but must compute (potentially much) larger cuts that only contain xor-shared variables (such cuts always exist as xor-assumptions are xor-shared). And, if DEDUCE-FULL finds that $\phi_{\text{xor}} \wedge l_1 \wedge ... \wedge l_n$ is unsatisfiable, it must deduce and return a subset $\{l'_1, ..., l'_k\} \subseteq$

$\{l_1, ..., l_n\}$ such that $\phi_{\text{xor}} \models (\neg l'_1 \vee ... \vee \neg l'_k)$. These three facts mean that the SAT solver cannot learn anything about the internal structure of the xor-part but only about the xor-shared "interface" variables.

2. Treat the xor-internal variables as if they were xor-shared in the SAT solver. As they do not occur in the or-clauses, they are probably not assigned in the beginning of the SAT solver search but only when there are no real xor-shared variables left. As opposed to the previous approach, this approach has the advantage that the closest cnf-compatible cuts can be used when performing EXPLAIN and the SAT solver can learn about the interaction of the xor-internal variables by storing the or-clauses returned by EXPLAIN. However, if the or-clauses returned by the EXPLAIN method are all stored by the SAT solver, it essentially performs a cnf-translation to the xor-part ϕ_{xor}; we consider this undesirable and only store such clauses if they were needed during a conflict analysis performed by the SAT solver, the goal being to let the SAT solver learn only about the parts of ϕ_{xor} that were difficult. Of course, such stored clauses are subject to usual removal ("forgetting") heuristics applied in SAT solvers to avoid memory congestion.
3. Eliminate all the xor-internal variables in a preprocessing step by substituting them with their "definitions"; e.g. if $x_1 \oplus x_2 \oplus x_3$ is a xor-clauses with a xor-internal variable x_1, then remove the clause and replace every occurrence of x_1 in all the other xor-clauses by $x_2 \oplus x_3 \oplus \top$. This approach is simple to implement but has the drawback that the xor-clauses tend to grow longer, making our proof system less effective. As an example, assume the xor-clauses $(x \oplus a \oplus z)$, $(y \oplus d \oplus z)$, and $(z \oplus b \oplus c)$, where z is xor-internal; now the equivalence $x \oplus y \oplus \top$ can be deduced from the xor-assumptions (a) and (d). If z is eliminated, we have the xor-clauses $(x \oplus a \oplus b \oplus c \oplus \top)$ and $(y \oplus d \oplus b \oplus c \oplus \top)$; now the equivalence $x \oplus y \oplus \top$ cannot be deduced from (a) and (d).

Due to the reasons discussed above and some preliminary experimental evaluations, we have chosen to use the second approach in the experiments in the next section.

4 Experimental Results

We have evaluated the efficiency of the proposed DPLL(XOR) approach by implementing the xor-module and integrating it to the minisat [7] (version 2.0 core) solver in the way described in the previous section.[2] As benchmarks we consider instances that contain a large number of xor-clauses in order to show cases where the SAT solver enhanced with XOR reasoning outperforms the unmodified solver, but also problem instances that have only a few equivalences/XORs to demonstrate that enabling XOR reasoning does not hinder the SAT solver's performance in cases where it cannot reduce the number of decisions. The three benchmark families we consider are: known plain text attack on the block cipher DES, randomly generated linear problems based on 3-regular bipartite graphs, and known keystream attack on the stream cipher Trivium. All tests were run on Linux machines with 2GHz Intel Xeon 5130 processors; the available memory was limited to four gigabytes and time to four hours. The cipher attack benchmark instances were generated by first modelling the cipher and the attack as a Boolean circuit and then converting the Boolean circuit into (i) the standard DIMACS CNF format, and (ii) a DIMACS-like format allowing xor-clauses as well.

[2] Further experimental results and other ways to compute explanations and integrate the xor-reasoning module into a SAT solver can be found in [11].

Block Cipher DES. We modelled known plain texts attacks on two configurations of DES: 3 rounds with 1 block, and 4 rounds with 2 blocks. On these instances only around 1% of the clauses are xor-clauses and the xor-clauses are furthermore partitioned into small clusters separated by large number of or-clauses (here a cluster means a non-empty, minimal subset of xor-clauses such that if two xor-clauses share a variable, then they belong to the same cluster). Therefore, we did not expect that the xor-module would help in pruning the search space on these instances but included them as a "sanity check" to show that xor-reasoning does not degrade the performance too much on these kinds of instances. When compared to the standard minisat, the results indicate that there is a small overhead in the number of heuristic decisions required to solve the problems and a small constant time overhead in run time (most probably caused by not yet fully optimized data structures in xor-minisat).

Randomly Generated Regular Linear Problems. As a second test case we studied the impact of the xor-module on solving artificial problem instances consisting only of xor-clauses based on random generated 3-regular bipartite graphs presented in [9]. As these instances are fully linear, they can effectively be solved as-such by Gaussian elimination, so we considered modified instances obtained by converting a random amount of the XOR expressions to CNF. The problem instances included satisfiable and unsatisfiable instances with a number of variables ranging from 96 to 240. When compared to unmodified minisat, applying the xor-module on these highly regular, random problems did not reduce the number of heuristic decision required. We suspect that this is due to the random nature of the instances, which typically is not well suited for clause learning techniques applied also in our approach. Moreover, due to the strong regularity of the problem instances, there may be fewer cases where the equivalence inference rules can be used to prune the search space.

Stream Cipher Trivium. The most interesting benchmark we studied was a "known key-stream attack" on the stream cipher Trivium. The attack is modelled by generating a small number (from one to twenty in our experiments) of keystream bits after the 1152 initialization rounds. The 80-bit initial value vector is randomly generated and given in the problem instance. The 80-bit key, however, is left open. As there are far fewer generated keystream bits than key bits, a number of keys probably produce the same prefix of the keystream. Thus, the instances we benchmarked are all satisfiable. Structurally these instances are very interesting for benchmarking xor-reasoning as they have a large number of xor-clauses and these xor-clauses are also tightly connected so that there are only two or three large xor-clusters in each instance. For example, on instances with three xor-clusters, each xor-cluster has 2600–2900 xor-clauses involving 3500–3800 variables, while the or-part contains typically 8000–8600 or-clauses involving 5250–5700 variables.

In these experiments we compare our xor-minisat to the unmodified minisat, eqsatz [12] version 20, and march_hi[3] which is an optimized version of march_eq [10]. In the experiments, we ran twenty instances of each keystream length from one to twenty with each solver. The results are shown in Fig. 4; the dots on vertical and horizontal lines denote runs that exceeded the time limit.

When compared to the unmodified minisat, xor-minisat needs to make much less heuristic decisions when solving the instances. This is as expected because the underlying proof system in the xor-module includes equivalence reasoning in addition to unit propaga-

[3] Available at http://www.st.ewi.tudelft.nl/sat/Sources/sat2009/march_hi.zip

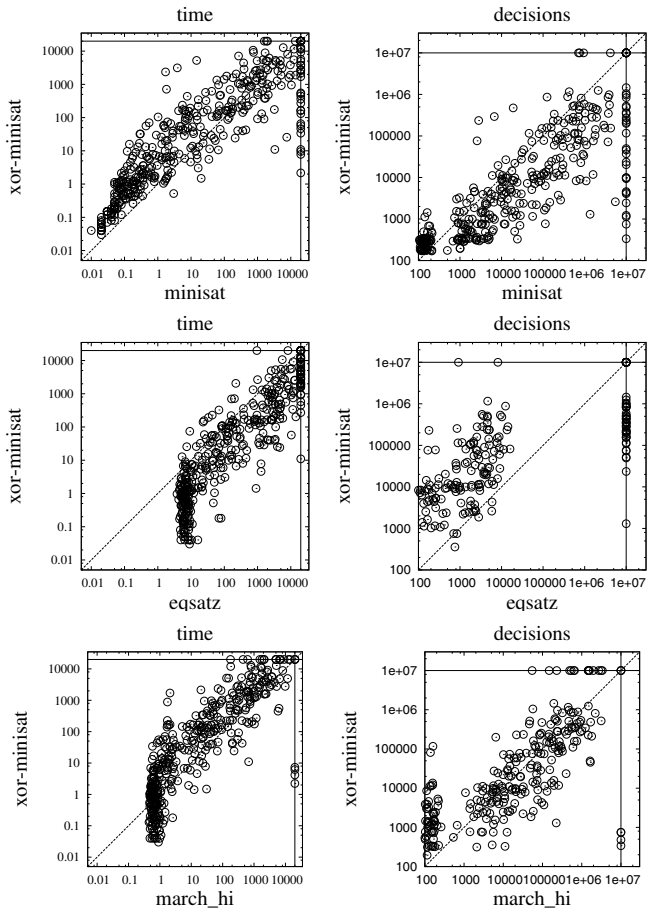

Figure 4. Results on the trivium benchmark set.

tion. However, if we compare the run times especially on the easier instances, it is clearly visible that the data structures and algorithms for xor-reasoning in xor-minisat are not yet as fully optimized as those for standard Boolean reasoning in minisat. On harder instances the stronger deduction system then helps to compensate this and the run times become comparable and even better on very hard instances.

Comparing xor-minisat to eqsatz we observe that the lookahead technique applied in eqsatz is a very strong deduction method as eqsatz makes even less decisions than xor-minisat. However, computing the lookahead is also very time consuming and, thus, eqsatz performs much worse when run time is considered.

When comparing against march_hi, the following observations can be made. First, when compared to eqsatz, march_hi seems to trade deduction power for speed as it does not suffer from the same run time penalty but neither prunes the search space as efficiently with (partial) lookahead and the other applied techniques (such as problem preprocessing, adding resolvents as well as finding and using binary xor-clauses). As a result, xor-minisat explores slightly smaller search spaces than march_hi on moderately hard and hard problems but needs to be further optimized to reach similar run times.

As a summary of these results, we can say that the proof system used in the xor-module as well as the way it has been integrated with a conflict-driven clause learning SAT solver are together indeed effective in reducing the size of the search space. However, the data structures and algorithms in xor-minisat still seem to need some optimization as xor-minisat appears to suffer from some constant factor inefficiencies. Furthermore, some preprocessing techniques should definitely be developed for xor-minisat.

5 Conclusions

The paper considers a combined satisfiability problem where the input consists of or-clauses and xor-clauses. A novel DPLL(T) style approach to integrating xor-reasoning to a SAT solver has been developed based on a xor-reasoning module which offers implied literals detection and clausal explanations. The module can be straightforwardly integrated into a state-of-the-art conflict-driven clause learning SAT solver, enabling clause learning over the combination of or-clauses and xor-clauses.

We have developed a prototype implementation of the xor-reasoning module and integrated it into a state-of-the-art conflict-driven SAT solver minisat. The implementation has been evaluated using challenging test cases involving combinations of or- and xor-clauses. The results are encouraging as, in particular, the number of decisions typically decreases if xor-clauses are exploited directly when compared to translating them to CNF. Also run times are comparable, especially on some hard instances.

REFERENCES

[1] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani, 'A SAT based approach for solving formulas over boolean and linear mathematical propositions', in *Proc. CADE 2002*, volume 2392 of *LNCS*, pp. 195–210. Springer, (2002).

[2] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, 'Satisfiability modulo theories', in *Handbook of Satisfiability*, IOS Press, (2009).

[3] P. Baumgartner and F. Massacci, 'The taming of the (x)or', in *Proc. CL 2000*, volume 1861 of *LNCS*, pp. 508–522. Springer, (2000).

[4] A. Biere, M. Heule, H. van Maaren, and T. Walsh, eds. *Handbook of Satisfiability*. IOS Press, 2009.

[5] J. Chen, 'Building a hybrid SAT solver via conflict-driven, look-ahead and XOR reasoning techniques', in *Proc. SAT 2009*, volume 5584 of *LNCS*, pp. 298–311. Springer, (2009).

[6] L. de Moura and H. Rueß, 'Lemmas on demand for satisfiability solvers', in *Proc. SAT 2002*, (2002).

[7] N. Eén and N. Sörensson, 'An extensible SAT solver', in *Proc. SAT 2003*, volume 2919 of *LNCS*, pp. 502–518. Springer, (2004).

[8] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, 'DPLL(T): Fast decision procedures', in *Proc. CAV 2004*, volume 3114 of *LNCS*, pp. 175–188. Springer, (2004).

[9] H. Haanpää, M. Järvisalo, P. Kaski, and I. Niemelä, 'Hard satisfiable clause sets for benchmarking equivalence reasoning techniques', *J. Satisfiability, Boolean Modeling and Computation*, **2**(1-4), 27–46, (2006).

[10] M. Heule, M. Dufour, J. van Zwieten, and H. van Maaren, 'March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver', in *Proc. SAT 2004*, volume 3542 of *LNCS*, pp. 345–359. Springer, (2004).

[11] T. Laitinen, 'Extending sat solvers with parity constraints', Research Report TKK-ICS-R32, Aalto University, Department of Information and Computer Science, (2010).

[12] C. M. Li, 'Integrating equivalency reasoning into Davis-Putnam procedure', in *Proc. AAAI/IAAI 2000*, pp. 291–296. AAAI Press, (2000).

[13] J. Marques-Silva, I. Lynce, and S. Malik, 'Conflict-driven clause learning SAT solvers', in *Handbook of Satisfiability*, IOS Press, (2009).

[14] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, 'Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)', *J. ACM*, **53**(6), 937–977, (2006).

[15] R. Ostrowski, É. Grégoire, B. Mazure, and L. Sais, 'Recovering and exploiting structural knowledge from CNF formulas', in *Proc. CP 2002*, volume 2470 of *LNCS*, pp. 185–199. Springer, (2002).

[16] M. Soos, K. Nohl, and C. Castelluccia, 'Extending SAT solvers to cryptographic problems', in *Proc. SAT 2009*, volume 5584 of *LNCS*, pp. 244–257. Springer, (2009).