XOR Satisfiability Solver Module for DPLL Integration

Tero Laitinen

April 7, 2010

Abstract

Satisfiability solvers that are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm operate on propositional logic formulas in conjunctive normal form (CNF). Despite major improvements in solver technology, using only CNF does not seem to scale well for problem instances involving XOR expressions.

We present a decision procedure to determine effectively the satisfiability of XOR clauses and to extract an explanation from conflicting assignments. Our modular proof-of-concept implementation can be integrated tightly into a DPLL-based solver capable of conflict-driven clause learning.

1 Introduction

The Boolean Satisfiability problem (SAT) is to decide whether a given Boolean formula has a model, that is, a truth assignment which assigns a truth value for all the propositional variables in the formula in such a way that the resulting formula evaluates to true. The SAT problem is an NP-complete problem, one of the hardest problems whose solutions can be verified in polynomial time, i.e. using a polynomial number of steps with regard to the size of the problem description. [6]

If there is an efficient algorithm for a complete problem of a complexity class (such as NP), the algorithm can be used to solve any problem in the same complexity class by means of reductions. A reduction translates an instance of a problem into an instance of another problem. A reduction has to be computationally feasible in order to be useful. Often a reduction of a new problem to a known problem is easier to implement than a specific algorithm for the new problem. [13]

The SAT problem can be solved in exponential time in the worst case but real world problems tend to be structured in a way that can be exploited to produce a solution even when the problem instances involve hundreds of thousands of variables. A satisfiability solver is a computer program that takes a SAT problem instance as its input and outputs a model if one exists. If the instance has no solutions, the solver terminates and reports that no solutions exist. The Davis-Putnam-Logemann-Loveland (DPLL) algorithm family [7] is a widely used decision procedure for SAT problem instances in conjunctive normal form (CNF).

SAT solvers and problem reductions (to conjunctive normal form) are not only of theoretical interest. For instance, symbolic model checking is an important problem domain of verifying that a hardware/software implementation satisfies a specification in cases where a device/program cannot be exhaustively tested by traditional means. General-purpose SAT solvers were shown to outperform some tools specially designed for model checking in a bounded environment where only counterexamples (a violation of a specification) of some maximum length were considered [3]. Other applications areas where SAT solvers have been successfully used include planning and verification problems, see e.g. chapters 15 and 16 in [4].

However, some problems cannot be solved efficiently when converted to CNF. Problem domains such as logical cryptanalysis and circuit verification are intractable for SAT solvers when solving larger instances. Experimental results suggest that extensive use of certain simple logical operators - such as equivalence operator ↔ or its negative counterpart, the exclusive or (XOR) - yields problem descriptions in CNF that current state-of-the-art SAT solvers scale poorly with [2].

There have been several improvements in the theory of solving SAT problems since the basic DPLL decision procedure was published. Along with these discoveries the performance of SAT solvers has increased drastically. One of the most important techniques is called conflict-driven learning [15][19]. It enhances the backtracking search by adding clauses when a conflict (a logical inconsistency) is reached. These clauses prune a part of the search space that is known not to contain any solution. The majority of current SAT solvers use some form of conflict learning.
There have been attempts to integrate equivalence reasoning in a SAT solver to tackle better with problem instances involving equivalences. EqSatz by Li \cite{11} 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 by Heule and van Maaren \cite{8} extracts equivalences from a CNF formula and produces a minimum solution for the linear part before starting the actual DPLL-based search procedure. The equivalences extracted in the pre-processing phase are taken into account when selecting which literal to branch on. Binary equivalences are kept in CNF due to optimized data structures. Unary clauses are used to simplify the set of equivalences which in turn may produce more binary equivalences. The solver MoRSat by Chen \cite{5} extracts equivalences from CNF, too. It uses a $2 \times 2$ watched literal scheme to do unit propagation effectively on XOR constraints. CryptoMiniSat by Soos et al. \cite{14} accepts a mixture of CNF and xor-clauses as its input. The xor-clauses are like ordinary or-clauses in the solver’s data structures but change appearance according to the assigned literals. In addition to more compact representation of xor-clauses it performs Gaussian elimination after a specified number of literals have been assigned and no other propagation rules can be fired. According to the experimental results, the equivalence reasoning techniques contribute to the overall performance of all solvers.

A generalization of the SAT problem is called Satisfiability Modulo background Theories (SMT) problem where the propositional atoms are not necessarily plain Boolean variables but first-order logic expressions, too. A background theory defines how predicate and function symbols are interpreted. The interpretations can then be used to reason about the logical consistency of truth assignments. \cite{12}

Accordingly, a program that solves the SMT problem is called SMT solver. SMT solvers can be implemented in a modular way so that modules correspond to solvers for background theories. Modules are integrated then into an SMT solver through a simple interface. A solver for a more specific theory - for example, of arithmetic integer expressions - can compute logical consistency of a problem instance more efficiently compared to a SAT solver when the problem instance is represented as a CNF formula. Such a solver module can use better suited data types, operate on problem instances at a higher level of abstraction and exploit the semantics of the theory. This kind of optimization is not possible when the problem is represented in Boolean domain. An SMT solver contains a SAT solver in the core. The SAT solver assigns truth values to the visible atoms of the theory modules, for example ($x > y$). The module then checks whether the set of atoms is consistent and notifies the SAT solver. If the theory module detects an inconsistency, a explanation for the conflict is returned to the SAT solver. This enables the SAT solver to accumulate information about relationships of the atoms while the search progresses. Based on the information the SAT solver has “learned” so far, some assignments are undone and the search continues possibly more accurately towards a potential solution.

This paper studies how to facilitate solving problems that can be formulated succinctly as a combination of CNF and XOR expressions. Our work focuses on defining a subroutine for deciding the satisfiability of XOR expressions and extracting information that can be used to prune the search space of the SAT solver which operates the subroutine.

We develop a decision procedure to determine the satisfiability of XOR expressions involving propositional variables and a modular implementation which can be integrated in a SAT solver. As variables are Boolean, the extended expressivity of first-order logic is not needed. Thus, there is no need for separate variables for representing visible atoms of the XOR theory but variables can be shared with the SAT solver. As XOR expressions effectively correspond to a system of linear equations modulo 2, the most straightforward way to solve XOR expressions is by Gaussian elimination (GE) \cite{17}, an algorithm that solves such systems with time complexity of $O(n^3)$ where $n$ is the number of equations and variables. Our preliminary tests, where we simulated DPLL-style backtracking search, suggest that GE is computationally expensive to carry out iteratively. That is why we decided to apply only a few simple inference rules during the DPLL search procedure. However, if the XOR expressions involve propositional variables that do not have occurrences in the CNF formula, GE is used to assign values for the remaining free variables after all variables of the CNF formula have been assigned by the SAT solver. Feasibility of applying (possibly partial) GE during the search remains to be studied in depth. When a conflict is reached, an explanation for the conflict is derived by consulting a graph of inferred xor-clauses. The explanation is always a single disjunction of literals in order to ease the integration of the module into a SAT solver.

Our approach differs from the equivalence reasoning procedures of EqSatz and march_eq in how equivalences are represented. Instead of pattern matching equivalences from the CNF formula where the equivalences converted to CNF can be substantially bigger than an optimal representation of the equivalences, the linear part of the problem description is presented as-is.
The paper is organized as follows. First some formal notation is presented. Then we give an overview of the purpose of the XOR module by presenting its interface. We continue by describing the inference rules. The soundness and completeness of the rules are discussed after this. A straightforward translation of XOR expressions to CNF formula is presented in order to show in the section that follows that the unit propagation inference rule can draw the same conclusions for logically equivalent XOR and CNF formulas. Also, the capabilities of equivalence reasoning rule are shown to be beyond those of plain unit propagation only. We introduce a concept of implication graph that we use to describe what is an explanation for a conflict and how such an explanation is deduced. We describe the key algorithms and core data structures of our proof-of-concept implementation. We present the results of an experimental test where we piloted the applicability of the approach with the implementation on randomized test instances. In the last section, we conclude the study and consider ideas for future work.

2 Preliminaries

We first introduce some notation that is used to describe the decision procedure and related proofs.

An atom is either a propositional variable or the symbol ⊤ which denotes the truth value True. A literal is an atom A or its negation ¬A. We use the symbol ⊥ as a shorthand to denote the literal ¬⊤. An or-clause is an expression $C = L_1 \lor \cdots \lor L_n$ where $L_1, \ldots, L_n$ are literals and the symbol $\lor$ stands for the non-exclusive logical OR. A conjunction of or-clauses is an expression $\phi = C_1 \land \cdots \land C_n$ where $C_1, \ldots, C_n$ are or-clauses and the symbol $\land$ stands for the logical AND. A clause is either an or-clause or a xor-clause.

A truth assignment $M$ is a set of atoms. It always holds that the atom $\top \in M$, i.e., $M \models \top$ where the operator $\models$ is read as “is a model for”. Let $A$ be an atom. We define that $M \models A$ iff $A \in M$. Given a literal $¬A$ it holds that $M \models ¬A$ iff $A \notin M$. A literal $L$ is said to be satisfied in $M$ iff $M \models L$. Let $C = L_1 \lor \cdots \lor L_n$ be an or-clause. It holds that $M \models L_i$ for all clauses $C \models L_i$ iff there is a literal $L_i$ in $C$ for which $M \models L_i$. Let $\phi = C_1 \land \cdots \land C_n$ be a CNF formula. It holds that $M \models \phi$ iff for all $C_i \in \phi$ it holds that $M \models C_i$. Let $X = L_1 \oplus \cdots \oplus L_n$ be a xor-clause. It holds that $M \models L_i$ for all clauses $C_i \models L_i$ iff there is an odd number of satisfied literals in $X$. The negated operator $\not\models$ is read as “is not a model for”. For anything $P$ for which $M \not\models P$ is defined, we define that $M \models P$ holds iff $M \models P$ does not hold.

Let $\Sigma_a$ and $\Sigma_b$ be sets of clauses (interpreted as conjunctions). The set $\Sigma_a$ is a logical consequence of the set $\Sigma_b$, denoted by $\Sigma_a \models \Sigma_b$, iff for all truth assignments $M$ the following holds: if for all clauses $C_a \in \Sigma_a$ it holds that $M \models C_a$, then for all clauses $C_b \in \Sigma_b$ it holds that $M \models C_b$. The right hand side set $\Sigma_b$ can be written without set braces $\{\}$ when there is exactly one element in the set.

Let $A$ be an atom different from $\top$ and $X$ be a xor-clause. A simplification of a xor-clause $C$ is denoted by $C[A/X]$ which defines a new xor-clause identical to $C$ except that all occurrences of $A$ are substituted with $X$.

2.1 Translation to Normal Form

In the case of xor-clauses, the negation can be eliminated in such a way that its only occurrence is in the xor-clause $\bot$. This simplifies the definition of the rules of the proposed decision procedure. Also, a xor-clause in the general case may contain redundancy (multiple or unnecessary literals) which can be removed while preserving models of the xor-clause.

Definition 1. A xor-clause in is normal form if it is either (i) $\bot$ or (ii) a xor-clause that contains only atoms and no atom appears twice.

The rewrite rules in Figure 1 for transforming xor-clauses to the normal form are based on the rules presented in [2]. The left hand side is the premise, the right hand side is the conclusion, $A$ is an atom and $C$ is a xor-clause. Provided that xor-clauses with the same atoms regardless of the order are considered the same, each xor-clause has exactly one corresponding xor-clause in normal form so the rules can be applied in any order. For instance, the normal form of $¬X \oplus Y \oplus Z \oplus Z$ is $X \oplus Y \oplus \top$, where $X$, $Y$ and $Z$ are propositional variables.

Let $C$ and $X$ be xor-clauses and $A$ be an atom. In the remainder of the paper, we will assume that all xor-clauses of the type $C[A/X]$ are implicitly transformed to normal form.
\[ \bot \oplus C \leadsto C \]
\[ \neg A \oplus C \leadsto A \oplus \top \oplus C \]
\[ A \oplus A \oplus C \leadsto C \]
\[ A \oplus C \oplus A \leadsto C \]
\[ A \oplus A \leadsto \bot \]

Figure 1: Rewrite rules for transforming xor-clauses to normal form

### 3 Overview of XOR module design

In this section we briefly present what kind of problems we intend to solve, what is a solution for such problems and how a solution is searched. The general setting will hopefully clarify the design decisions documented in the rest of the paper.

A problem instance consists of a set of or-clauses \( \Sigma_{\text{cnf}} \) and a set of xor-clauses \( \Sigma_{\text{xor}} \). A solution to the problem is a truth assignment \( M \) such that \( M \models \Sigma_{\text{cnf}} \) and \( M \models \Sigma_{\text{xor}} \). We aim to extend the functionality of a SAT solver by a XOR module that implements the interface in Table 1. The CNF part \( \Sigma_{\text{cnf}} \) is given to the SAT solver and the XOR part \( \Sigma_{\text{xor}} \) to the XOR module. The interface and the implementation are discussed in more detail in Section 8.

The normal DPLL-based search on \( \Sigma_{\text{cnf}} \) is extended by propagating assignments \( l_1, \ldots, l_n \) to the XOR module when appropriate where \( l_1, \ldots, l_n \) are literals that fix values of some of the shared variables (variables that occur both in \( \Sigma_{\text{cnf}} \) and in \( \Sigma_{\text{xor}} \)). If the XOR module can deduce that the partial truth assignment cannot be a subset of any solution, i.e., the formula \( F = \Sigma_{\text{xor}} \land l_1 \land \cdots \land l_n \) does not have any models, it signals the SAT solver and supplies an explanation why the formula \( F \) is unsatisfiable. The explanation is an or-clause \( C \) such that \( \Sigma_{\text{xor}} \models C \) and \( \Sigma_{\text{xor}} \cup (l_1, \ldots, l_n) \models \neg C \). The SAT solver analyzes the explanation, stores some information about the inconsistency to avoid similar fruitless situations in further search and asks the XOR module to cancel some of the assignments by restoring one of the explicitly recorded previous states of the XOR module.

The process is repeated until a solution is found or it can be deduced that no such solution exists. In the following sections we focus in describing the internals of the XOR module.

### 4 Inference Rules

In this section we define the rules of the decision procedure and establish the soundness of the rules.

We use the following notation to describe a rule:

\[
\begin{array}{c|c|c}
\text{Name} & C_i & C_k \\
\hline
\text{Introduce} & C_j & C_i \\
\end{array}
\]

where \( C_i, C_j \) and \( C_k \) are xor-clauses. Given \( C_i \) and \( C_j \), the application of the rule results in a new xor-clause \( C_k \). The new xor-clause \( C_k \) is always implicitly transformed to normal form.

The Introduce-rule is used to add a xor-clause \( C \) from a set of xor-clauses \( \Sigma \). Its counterpart in the implementation is the method \texttt{ADD-CLAUSE} defined in Table 1. The xor-clauses added using this rule define the problem instance.

\[
\text{Introduce} \quad C
\]

The rules Decide\(^+\) and Decide\(^-\) are in fact special cases of Introduce-rule. As with Introduce-rule, any xor-clause added with these rules has to be in the set \( \Sigma \). Applications of these rules correspond to invocations of the method \texttt{ASSIGN} defined in Table 1. The rules are defined to distinguish the assignments done by the SAT solver, which is needed when we define how an explanation for conflicting assignments (the return value of the method \texttt{EXPLAIN}) is calculated.
### Table 1: Interface of LibXorSat module

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD-CLause</td>
<td>declares a new xor-clause</td>
<td>identifier of the xor-clause</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>assigns a truth value to a variable</td>
<td>false, if the set of xor-clauses is no longer satisfiable, true otherwise</td>
</tr>
<tr>
<td>BACKJUMP</td>
<td>restores the state recorded by a previously added backjump point</td>
<td>-</td>
</tr>
<tr>
<td>EXPLAIN</td>
<td>analyzes the conflict</td>
<td>an or-clause which is a logical consequence of the added xor-clauses</td>
</tr>
<tr>
<td>REMOVE-CLause</td>
<td>removes a previously added xor-clause</td>
<td>-</td>
</tr>
<tr>
<td>REMOVE-VARIABLE</td>
<td>removes a previously added variable</td>
<td>-</td>
</tr>
<tr>
<td>RESET</td>
<td>cancels all assignments</td>
<td>-</td>
</tr>
<tr>
<td>SET-BACKJUMP</td>
<td>adds a backjump point which records the state of the solver</td>
<td>identifier of the backjump point</td>
</tr>
<tr>
<td>SOLVE</td>
<td>performs full Gaussian elimination</td>
<td>false, if the set of xor-clauses is no longer satisfiable, true otherwise</td>
</tr>
</tbody>
</table>

The rules Introduce, Decide $^+$ and Decide $^-$ are referred as New-rules.

The Inference-rules of the proof system are listed in Figure 2, where $A, A_1$ and $A_2$ are atoms different from $\top$ and $C$ is a xor-clause. The rules are adopted from the paper by Baumgartner and Massacci [2].

$$
\begin{align*}
\oplus \text{-Unit}^+ & : \quad \frac{A \quad C}{C [A / \top]} \\
\oplus \text{-Unit}^- & : \quad \frac{A \oplus \top \quad C}{C [A / \bot]} \\
\oplus \text{-Eqv}^+ & : \quad \frac{A_1 \oplus A_2 \oplus \top \quad C}{C [A_1 / A_2]} \\
\oplus \text{-Eqv}^- & : \quad \frac{A_1 \oplus A_2 \quad C}{C [A_1 / (A_2 \oplus \top)]}
\end{align*}
$$

**Figure 2: Inference rules**

**Definition 2.** Let $\Sigma$ be a set of xor-clauses. A xor-derivation from $\Sigma$ is a sequence of xor-clauses $D = C_1, \ldots, C_n$ where each $C_k$, $1 \leq k \leq n$, is either added using one of New-rules or is derived from two xor-clauses $C_i$ and $C_j$, $i < k$ and $j < k$ using one of Inference-rules.

A xor-derivation is a xor-refutation if the last xor-clause in the xor-derivation is $\bot$.

**Example 1.** An example of a xor-refutation is shown in Figure 3. Note that the symbol $\top$ is always moved to the end of xor-clause and any intermediate steps for transforming xor-clauses to normal form are omitted, for instance, at the step 9: $(x_2 \oplus x_3 \oplus \top) [x_2 / \top] = \top \oplus x_3 \oplus \top = x_3$.

New-rules and Inference-rules rules define a sound and incomplete proof system. Here, the soundness of the proof system means that if there is a xor-derivation $D = C_1, \ldots, C_n$ from a set of xor-clauses $\Sigma$, then it holds that $\Sigma \models C_n$. By incompleteness we mean that given a set of xor-clauses $\Sigma$, if there is a xor-clause $C$ such that $\Sigma \models C$, it does not generally hold that there is a xor-derivation $D = C_1, \ldots, C_n$ from $\Sigma$ such that $C_n = C$. The soundness of rules is covered by Theorem 1. The limitations of Inference-rules, i.e., the incompleteness is illustrated in Example 2.

The proof system is refutation-complete for sets of xor-clauses where each set of xor-clauses $\Sigma$ has a xor-clause of type $A$ or $A \oplus \top$ for each variable $A$ occurring in xor-clauses of $\Sigma$ in the following sense: if the set $\Sigma$ does not have any models, then a xor-refutation can be constructed from $\Sigma$ by adding first all clauses from the set $\Sigma$ and then applying the inference rules $\oplus\text{-Unit}^+$ and $\oplus\text{-Unit}^-$ until $\bot$ is derived.
1. \( x_1 \oplus x_2 \oplus x_3 \) Introduce
2. \( x_1 \oplus x_2 \oplus x_3 \oplus x_5 \) Introduce
3. \( x_3 \oplus x_4 \oplus x_5 \) Introduce
4. \( x_1 \) Decide*
5. \( x_2 \oplus x_3 \oplus \top \) \( \oplus \)-Unit*(1, 4)
6. \( x_2 \oplus x_4 \oplus x_5 \oplus \top \) \( \oplus \)-Unit*(2, 4)
7. \( x_3 \oplus x_4 \oplus x_5 \oplus \top \) \( \oplus \)-Eqv*(5, 6)
8. \( x_2 \) Decide*
9. \( x_3 \) \( \oplus \)-Unit*(5, 8)
10. \( x_4 \oplus x_5 \oplus \top \) \( \oplus \)-Unit*(3, 9)
11. \( x_4 \oplus x_5 \) \( \oplus \)-Unit*(7, 9)
12. \( \bot \) \( \oplus \)-Eqv*(10, 11)

Figure 3: Example derivation of xor-clauses

Example 2. Let \( \Sigma \) be \( \{ (x_1 \oplus x_2 \oplus x_3), (x_1 \oplus x_2 \oplus x_3 \oplus \top) \} \). There is clearly no truth assignment \( M \) such that \( M \models \Sigma \). This implies \( \Sigma \not\models \bot \). Yet none of the inference rules can be used derive \( \bot \) from \( \Sigma \). Thus, the proof system is incomplete. The xor-derivation in Figure 4 illustrates a situation where none of Inference-rules is applicable.

1. \( x_1 \oplus x_2 \oplus x_3 \) Introduce
2. \( x_1 \oplus x_2 \oplus x_3 \oplus \top \) Introduce

Figure 4: Xor-derivation showing that the proof system is incomplete

Lemma 1. (soundness of Inference-rules)
Let \( C_i, C_j \) and \( C_k \) be xor-clauses. Assume that \( C_k \) is derived from \( C_i \) and \( C_j \) using one of the inference rules. It holds that \( \{ C_i, C_j \} \models C_k \).

Proof. Let \( \#(C, M) \) be the number of satisfied literals in the xor-clause \( C \) given the truth assignment \( M \). Let \( M \) be any truth assignment such that \( M \models \{ C_i, C_j \} \). Let \( A, A_1 \) and \( A_2 \) be atoms different from \( \top \).

\( \oplus \)-Unit*: Let \( C_j = A \). It holds that \( A \in M \) and \( M \models C_j \). As \( A \in M \) it can be substituted with \( \top \) without affecting the number of satisfied literals: \( \#(C_j, M) = \#(C_j[A/\top], M) \). It follows that \( \{ A, C_j \} \models C_j[A/\top] \).

\( \oplus \)-Unit*: Let \( C_i = A \oplus \top \). It holds that \( A \notin M \) and \( M \models C_j \). As \( A \notin M \) it can be substituted with \( \bot \) without affecting the number of satisfied literals: \( \#(C_j, M) = \#(C_j[A/\bot], M) \). It follows that \( \{ C_i, C_j \} \models C_j[A/\bot] \).

\( \oplus \)-Eqv*: Let \( C_i = A_1 \oplus A_2 \oplus \top \). It holds that \( A_1 \in M \) iff \( A_2 \in M \). \( A_1 \) can be substituted with \( A_2 \) in \( C_j \) without affecting the number of satisfied literals: \( \#(C_j, M) = \#(C_j[A_1/A_2], M) \). It follows that \( \{ A_1 \oplus A_2 \oplus \top, C_j \} \models C_j[A_1/A_2] \).

\( \oplus \)-Eqv*: Let \( C_i = A_1 \oplus A_2 \). It holds that \( A_1 \in M \) iff \( A_2 \notin M \). \( A_1 \) can be substituted with \( A_2 \oplus \top \) in \( C_j \) without affecting the number of satisfied literals: \( \#(C_j, M) = \#(C_j[A_1/(A_2 \oplus \top)], M) \). It follows that \( \{ A_1 \oplus A_2, C_j \} \models C_j[A_1/(A_2 \oplus \top)] \).

\( \square \)

Theorem 1. (soundness of xor-derivations)
Let \( D = C_1, \ldots, C_n \) be a xor-derivation from a set of xor-clauses \( \Sigma \). For each \( C_i, 1 \leq i \leq n \) it holds that if \( C_i \) was derived using one of Inference-rules, then \( \{ C_1, \ldots, C_{i-1} \} \models C_i \). Moreover, it holds that \( \Sigma \models C_i \).
Proof. We prove the theorem by induction. Base case $1 \leq i \leq 2$: $C_1$ and $C_2$ cannot be derived using one of Inference-rules because each rule needs two xor-clauses as premises, so the claim holds for $i \in \{1, 2\}$. Suppose the claim holds for each $m \in \{1, \ldots, i-1\}$. If $C_i$ was not derived using one of Inference-rules, the claim holds for $i$ by definition. Otherwise, $C_i$ was derived from two xor-clauses $C_j, 1 \leq j < i$ and $C_k, 1 \leq k < i, k \neq j$. By Lemma 1 it holds that $\{C_j, C_k\} \models C_i$. Under the assumption that $\Sigma \models C_j$ and $\Sigma \models C_k$, it holds that $\Sigma \models C_i$. By definition of logical consequence the left hand set of xor-clauses $\{C_j, C_k\}$ can be augmented by additional clauses without compromising logical consequence. It follows that $\{C_1, \ldots, C_{i-1}\} \models C_i$. □

5 Translation to CNF

A xor-clause $C_{xor}$ in normal form can be converted to a logically equivalent conjunction of or-clauses by listing all truth assignments that involve only variables of $C_{xor}$ and are not models for $C_{xor}$. Then corresponding or-clauses $C_{or}$ are constructed for each selected truth assignment $M$. Each or-clause is of the form $C_{or} = L_1 \lor \cdots \lor L_n$ where $L_1, \ldots, L_n$ are literals such that for each literal $L \in C_{or}$ there is exactly one corresponding variable $A$. If $\top$ is in $C_{xor}$, then $L = A$ iff $A \in M$, otherwise $L = \neg A$ iff $A \in M$. All variables of $C_{xor}$ have a corresponding literal in $C_{or}$. The translation results in the conjunction of the constructed or-clauses. We name this translation $cnf$. The function $cnf$ translates a xor-clause $C_{xor}$ to a logically equivalent minimal CNF formula.

Example 3. The translation of the xor-clause $x_1 \oplus x_2 \oplus x_3$ to a conjunction of or-clauses is shown in Table 2. The table lists all truth assignments (first three columns) and the corresponding or-clause (the last column). The resulting conjunction of or-clauses is $(x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2 \lor x_3)$.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_1 \oplus x_2 \oplus x_3$</th>
<th>CNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$x_1 \lor x_2 \lor x_3$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$x_1 \lor \neg x_2 \lor \neg x_3$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\neg x_1 \lor x_2 \lor \neg x_3$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$\neg x_1 \lor \neg x_2 \lor x_3$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Example translation to CNF

For the following proofs, let $|C_{xor}|$ be the number of variables in $C_{xor}$.

Lemma 2. The resulting CNF formula $cnf(C_{xor})$ is logically equivalent to the original xor-clause $C_{xor}$.

Proof. There are $2^{|C_{xor}|}$ different truth assignments $M$ involving only variables in $C_{xor}$ and half of these have an even number of variables, i.e. $M \not\models C_{xor}$. There is one constructed or-clause $C_{or}$ for each truth assignment $M$ such that $M \not\models C_{xor}$. Each or-clause $C_{or}$ in $cnf(C_{xor})$ has a literal for each variable in $C_{xor}$. Each $C_{or}$ has $2^{|C_{xor}|-1}$ models and in addition one truth assignment $M$ that is not a model for $C_{or}$ nor for $C_{xor}$. There are $2^{|C_{xor}|-1}$ distinct or-clauses and each one has a corresponding distinct truth assignment $M$ such that $M \not\models C_{xor}$. For each truth assignment $M$ it holds that if $M \models C_{xor}$, then $M \not\models \phi$ because there is no or-clause $C_{or}$ such that $M \not\models C_{or}$, and if $M \not\models C_{xor}$, then $M \not\models \phi$ because there is exactly one or-clause $C_{or}$ such that $M \not\models C_{or}$.
Lemma 3. Let $C_{\text{xor}}$ be a xor-clause in normal form with $|C_{\text{xor}}|$ variables. If no extra variables are introduced, any CNF formula that is logically equivalent to $C_{\text{xor}}$ has at least $2^{C_{\text{xor}}-1}$ or-clauses that contain at least one literal for each variable in $C_{\text{xor}}$.

Proof. Let $\phi$ be a CNF formula that is logically equivalent to the xor-clause $C_{\text{xor}}$. Assume that the claim does not hold. The CNF formula $\phi$ then contains less than $2^{C_{\text{xor}}-1}$ or-clauses that have at least one literal for each variable in the xor-clause $C_{\text{xor}}$.

Suppose that $C_{\text{xor}}$ does not have an occurrence of the atom $\top$. Then a truth assignment with an odd number of variables is a model for $C_{\text{xor}}$. The xor-clause $C_{\text{xor}}$ has $|C_{\text{xor}}|$ variables and there are $2^{|C_{\text{xor}}|}$ different truth assignments on variables of $C_{\text{xor}}$. Exactly half of these have an even number of variables so there are $2^{|C_{\text{xor}}|-1}$ truth assignments on variables of $C_{\text{xor}}$ that are not models for $C_{\text{xor}}$. Let $M_1, \ldots, M_{2^{|C_{\text{xor}}|-1}}$ be the truth assignments on variables of $C_{\text{xor}}$ that are not models for $C_{\text{xor}}$. For each truth assignment $M_i$ of the $2^{|C_{\text{xor}}|-1}$ previously defined truth assignments there is a corresponding or-clause $C_i$ with $|C_{\text{xor}}|$ literals such that $M_i \not\models C_i$.

The CNF formula $\phi$ has less than $2^{|C_{\text{xor}}|-1}$ or-clauses that have at least one literal for each variable in the xor-clause $C_{\text{xor}}$ so there is an or-clause $C_m = l_1 \lor \cdots \lor l_{|C_{\text{xor}}|}$ that is one of the or-clauses $C_1, \ldots, C_{2^{|C_{\text{xor}}|-1}}$ and not in the CNF formula $\phi$. The corresponding truth assignment $M_m$ is not a model for the xor-clause $C_{\text{xor}}$ nor for the xor-clause $C_m$. Let $x_1, \ldots, x_{|C_{\text{xor}}|}$ be the variables of the xor-clause $C_{\text{xor}}$. The truth assignment $M_m$ can be constructed as follows: for each literal $l_i$ in the or-clause $C_m$ if $l_i$ is $x_i$, then $x_i \in M_m$; if $l_i$ is $\neg x_i$, then $x_i \not\in M_m$. It clearly holds that $M_m \not\models C_m$. Now we define a function $\text{flip}(M, x_i)$ where $M$ is a truth assignment and $x_i$ is a variable. The function $\text{flip}(M, x_i)$ defines a truth assignment $M'_{lx}$ that is identical to $M$ with one exception: $x_i \in M'_{lx}$ iff $x_i \not\in M$. The number of variables in the truth assignment $M'_{lx}$ differs by one so there is an odd number of variables in $M'_{lx}$. Let us define a set of truth assignments $S = \{ \text{flip}(M, x_i) | x_i \in C_{\text{xor}} \}$. The truth assignment $M_m$ is not a model for $C_{\text{xor}}$ so each truth assignment in the set $S$ is a model for $C_{\text{xor}}$. As the CNF formula $\phi$ is logically equivalent to $C_{\text{xor}}$, it holds that $M_m \not\models \phi$ and for each $M \in S$ it holds that $M \models \phi$. Each truth assignment in $S$ must be a model for each or-clause in the CNF formula $\phi$. So there must be an or-clause $C_{\text{or}}$ in the CNF formula $\phi$ such that $M_m \not\models C_{\text{or}}$ and also each truth assignment in the set $S$ must be a model for $C_{\text{or}}$. It follows that the or-clause $C_{\text{or}}$ has to have a literal for each variable in the xor-clause $C_{\text{xor}}$ so that each truth assignment in the set $S$ is a model for $C_{\text{or}}$. The or-clause $C_{\text{or}}$ is precisely $C_m$ which was assumed not to be in the CNF formula $\phi$. This is a contradiction.

Suppose that the xor-clause $C_{\text{xor}}$ has an occurrence of the atom $\top$. The proof proceeds as above but the words “even” and “odd” are exchanged.

We assumed that there is a xor-clause $C_{\text{xor}}$ in normal form that has a logically equivalent CNF formula $\phi$ with less than $2^{|C_{\text{xor}}|-1}$ or-clauses that have at least one literal for each variable in $C_{\text{xor}}$ and arrived to a contradiction. Hence, the claim holds for all xor-clauses in normal form.

□

6 Simulation of Unit-Propagation in CNF

The motivation here is to show that the application of the inference rules $\oplus$-Unit$^+$ and $\oplus$-Unit$^-$ for xor-clauses is at least as effective as unit propagation for an equivalent CNF formula, that is to say, the same conclusions can be deduced regardless of the representation of the problem.

The corresponding inference rules of unit propagation for CNF formulas are shown in Figure 5, where $A$ is an atom different from $\top$ and $\phi$ is a CNF formula.

\[
\begin{align*}
\text{CNF-Unit}^+ & : & & \frac{A \quad \phi}{\phi[A/\top]} & \quad \text{CNF-Unit}^- & : & & \frac{\neg A \quad \phi}{\phi[A/\bot]}
\end{align*}
\]

Figure 5: Unit propagation inference rules for CNF formulas

For the following proofs, let $C_i$ and $C_j$ be xor-clauses and $A$ be a propositional variable. Let $\phi_i = cnf(C_i)$ and $\phi_j = cnf(C_j)$ be minimal logically equivalent CNF formulas to xor-clauses $C_i$ and $C_j$, respectively.
Lemma 4. If a xor-clause \( C_k \) can be inferred from the xor-clauses \( C_i \) and \( C_j \) using unit-propagation rules for xor-clauses, then a CNF formula \( \phi_k \) that is logically equivalent to \( C_k \) can be inferred from \( \phi_i \) and \( \phi_j \) using unit-propagation rules for CNF.

Proof. In order to infer the xor-clause \( C_k \) from the xor-clauses \( C_i \) and \( C_j \) using unit-propagation rules for xor-clauses, one of the latter two has to have only one variable. Due to symmetry, it suffices to consider the cases \( C_i = A \) and \( C_j = A \oplus \top \). The logically equivalent CNF formula \( \phi_i \) is \( A \) in the first case and \( \neg A \) in the second case. It follows that a CNF formula \( \phi_k \) can be inferred from the CNF formulas \( \phi_i \) and \( \phi_j \) using unit-propagation rules for CNF formulas.

If \( C_i = A \) and \( \phi_i = A \), then \( C_k = C_j [A / \top] \) and \( \phi_k = \phi_j [A / \top] \). Let \( M \) be any truth assignment. It holds that \( M \models C_k \) iff \( M \models C_j \) and \( A \in M \). It also holds that \( M \models \phi_k \) iff \( M \models \phi_j \) and \( A \in M \). As \( C_j \) and \( \phi_j \) are logically equivalent, it holds that \( M \models C_j \) iff \( M \models \phi_j \). It follows that \( M \models C_k \) iff \( M \models \phi_k \).

If \( C_i = A \oplus \top \) and \( \phi_i = \neg A \), the proof is equal except that the truth assignments \( M \) such that \( M \models \phi_j \) and \( A \notin M \) are models for \( C_k \) and \( \phi_k \).

\[ \square \]

Lemma 5. If a CNF formula \( \phi_k \) can be inferred from the CNF formulas \( \phi_i \) and \( \phi_j \) using unit-propagation rules for CNF, then a xor-clause \( C_k \) that is logically equivalent to \( \phi_k \) can be inferred from \( C_i \) and \( C_j \) using unit-propagation rules for xor-clauses.

Proof. As in the proof of Lemma 4, we consider only the cases \( \phi_i = A \) and \( \phi_i = \neg A \). The logically equivalent xor-clauses are \( C_j = A \) and \( C_j = A \oplus \top \). The unit propagation rules for xor-clauses and the unit propagation rules for CNF formulas are both applicable. Thus, the CNF formula \( \phi_k \) can be inferred from \( \phi_i \) and \( \phi_j \), and the xor-clause \( C_k \) can be inferred from \( C_i \) and \( C_j \). The proof continues as in the proof of Lemma 4.

\[ \square \]

The inference rules \( \oplus \text{-Eqv}^+ \) and \( \oplus \text{-Eqv}^- \) contribute to the effectiveness of the proof system because there are unsatisfiable sets of xor-clauses that do not have xor-refutations when using only the unit propagation rules \( \oplus \text{-Unit}^+ \) and \( \oplus \text{-Unit}^- \) but have xor-refutations when the inference rules \( \oplus \text{-Eqv}^+ \) and \( \oplus \text{-Eqv}^- \) can be used. This is illustrated in Example 4.

Example 4. The xor-refutation in Figure 6 uses only the rules \( \oplus \text{-Eqv}^+ \) and \( \oplus \text{-Eqv}^- \). There is clearly no xor-refutation that uses only the unit propagation rules \( \oplus \text{-Unit}^+ \) and \( \oplus \text{-Unit}^- \).

1. \( x_1 \oplus x_3 \oplus x_4 \oplus x_5 \) Introduce
2. \( x_1 \oplus x_3 \oplus x_4 \oplus x_5 \oplus \top \) Introduce
3. \( x_3 \oplus x_4 \oplus \top \) Introduce
4. \( x_1 \oplus x_5 \) \( \oplus \text{-Eqv}^+(1, 3) \)
5. \( x_3 \oplus x_4 \) \( \oplus \text{-Eqv}^-(2, 4) \)
6. \( \perp \) \( \oplus \text{-Eqv}^-(3, 5) \)

Figure 6: Conflicting xor-derivation using only \( \oplus \text{-Eqv}^+ \) and \( \oplus \text{-Eqv}^- \)

7 Conflict Analysis

A SAT solver whose search procedure is based on conflict-driven clause learning assigns heuristically selected truth values to variables whose value has not been fixed when nothing can be inferred by unit propagation. A heuristically picked truth value can often result in a conflict (an unsatisfied or-clause of the CNF formula). When this happens, the SAT solver adds a conflict clause in its database of learned clauses. A conflict clause is an or-clause that is a logical consequence of the CNF formula and has literals for a subset of variables whose values has been fixed. The conflict clause prevents the SAT solver from selecting the same truth values for the variables that caused the conflict and in a favorable case can also prune a larger part of the search space. How a SAT solver constructs its own learned clauses is beyond the scope of this paper. For our purposes it
Next we define what is an explanation for a conflict and how such an explanation is extracted from an implication graph. The learned clauses are redundant in the sense that they are logical consequences of the original problem description.

In order to take advantage of conflict-learning capabilities of a SAT solver, we consider conflict analysis of xor-refutations. Here, an explanation for a conflict (a xor-refutation) is a set of the xor-clauses. Also, it should be possible to convert an explanation to a short CNF formula so that the potential performance gain caused by the reduced search space is greater than the overhead of the additional CNF formula. In this section we define how to extract an explanation for a xor-refutation that can be translated to CNF efficiently.

### 7.1 Implication Graph

Xor-derivation and Inference-rules are enough to define the decision procedure. However, the decision procedure does not state what is an explanation for a xor-refutation or how such an explanation is derived. Defining this by using xor-refutation only would be complex. That is why we define a more intuitive graph structure for describing the conflict analysis method. We will extend the definition of implication graph described in [19].

We define some supporting concepts for the definition of implication graph. A directed graph is a pair \( (V, E) \) where \( V \) is the set of nodes of the graph and \( E \subseteq V \times V \) defines which nodes are connected by an edge. A tuple \( (s, t) \in E \) is an edge from the source node \( s \) to the target node \( t \). A path is a sequence of nodes such that two consecutive nodes in the path are connected by an edge. A directed graph is acyclic (DAG) if it does not have a path from a node to the same node. Let \( D = C_1, \ldots, C_n \) be a xor-derivation, the function \( \text{rule}(C_i) \) gives the rule used to introduce or derive the xor-clause \( C_i \) in \( D \), the function \( \text{premisesOf}(C_i) \) gives the set \( \{ C_j, C_k \} \) if the xor-clause \( C_i \) is derived from \( C_j \) and \( C_k \) in \( D \) using one of the inference rule, and the function \( \text{numDecision}(C_i) \) gives the number of applications of Decide\(^+\) and Decide\(^-\) in \( D \) up to and including \( C_n \), i.e., the size of the set \( \{ C_m \mid C_m \text{ is in } D, m \leq i, \text{rule}(C_m) \in \{ \text{Decide}^+, \text{Decide}^- \} \} \).

**Definition 3.** Given a xor-derivation \( D = C_1, \ldots, C_n \), the implication graph derived from \( D \) is a labeled directed acyclic graph \( G = (V, E, r, d) \), where:

- \( V = \{ C_1, \ldots, C_n \} \) is the set of nodes
- \( E = \{ \langle C_a, C_b \rangle \mid C_a \in V, C_b \in V, C_a \in \text{premises}(C_b) \} \) is the set of edges
- \( r : V \rightarrow \{ \text{Introduce}, \text{Decide}^+, \text{Decide}^-, \text{@-Unit}^+, \text{@-Unit}^-, \text{@-Eqv}^+, \text{@-Eqv}^- \} \) is defined as follows:
  - \( r(C_i) = \text{rule}(C_i) \)
- \( d : V \rightarrow \mathbb{N} \) is defined as follows:

\[
  d(C_i) = \begin{cases}
    0 & \text{rule}(C_i) = \text{Introduce} \\
    \text{numDecision}(C_i) & \text{rule}(C_i) \in \{ \text{Decide}^+, \text{Decide}^- \} \\
    \max \{ d(C_k) \mid C_k \in \text{premises}(C_i) \} & \text{rule}(C_i) \notin \{ \text{Introduce}, \text{Decide}^+, \text{Decide}^- \}
  \end{cases}
\]

A node of an implication graph is a conflict node if it is the xor-clause \( \bot \).

The set of nodes \( V \) consists of xor-clauses of the xor-derivation \( D \). The graph has an edge from a xor-clause \( C_a \) to a xor-clause \( C_b \) iff \( C_b \) was derived from \( C_a \) using one of the inference rules. The function \( r \) assigns a rule label to each node. The rule label tells which rule was used to add the xor-clause of the node to the xor-derivation. The function \( d \) assigns a decision level to each node. The decision level of a node tells how many of its ancestor nodes has the rule label Decide\(^+\) or Decide\(^-\).

**Example 5.** The implication graph of the xor-derivation in Figure 8 is shown in Figure 7 where the notation for the contents of a node is \( \{ i, C_i \mid r(C_i), d(C_i) \} \), where \( C_i \) is the \( i \)th xor-clause in the xor-derivation \( D \). The cut lines (cut 1, cut 2, cut 3) can be ignored for now.

### 7.2 Extracting Conflict Set

Next we define what is an explanation for a conflict and how such an explanation is extracted from an implication graph. We will use the definition of a cut described in [19].
Definition 4. (cut). Given an implication graph \( G = (V, E, r, d) \), a (conflict) cut is a pair \( (V_a, V_b) \) such that \( V = V_a \cup V_b \) and \( V_a \cap V_b = \emptyset \). The first part of the cut \( V_a \) contains at least the nodes with no incoming edges and \( V_b \) contains at least the nodes with no outgoing edges.

A cut splits an implication graph into two parts where the first part justifies the second part. This implies that the justifying part of a cut contains the explanation for a conflict (provided that one exists). The justifying part of a cut may contain xor-clauses that are added using Introduce-rule and thus belong to the xor-part of the original problem instance. An explanation for a conflict is to be returned back to the SAT solver operating the XOR module (see method \text{explain} in Table \ref{table:explain}), so these xor-clauses are not included in the explanation because they are assumed valid (which is not the case with the xor-clauses added using Decide\(^+\) or Decide\(^-\)). Xor-clauses added using Introduce-rule can be excluded from an explanation for a conflict but additional xor-clauses can be removed, too. Only the necessary xor-clauses needed to derive a conflict should be included in an explanation.

Definition 5. Let \( D = C_1, \ldots, \perp \) be a xor-refutation, \( G = (V, E, r, d) \) be an implication graph of the xor-refutation \( D \) and \( (V_a, V_b) \) be a cut of \( G \). The conflict set of the cut \( (V_a, V_b) \) is a set of xor-clauses \( \Sigma_{conf} \) and the supporting set of the cut \( (V_a, V_b) \) is a set of xor-clauses \( \Sigma_{supp} \) where:

\[
\Sigma_{conf} = \{ s | s \in V_a, r(s) \neq \text{Introduce}, \exists t \in V_b : (s, t) \in E \} \\
\Sigma_{supp} = \{ s | s \in V_a, r(s) = \text{Introduce}, \exists t \in V_b : (s, t) \in E \}
\]

We take into consideration only the xor-clauses that "cross the cut boundary", that is, have direct successors in the part of the implication graph defined by the second part of a cut. By the definition of implication graph and the definition of cut, these are exactly the xor-clauses needed to derive the xor-clauses of the second part of a cut. This set of xor-clauses is again split into two sets: the conflict set and the supporting set. The supporting set is a subset of the xor-part of the original problem instance and the conflict set contains xor-clauses that (i) added using Decide\(^+\) or Decide\(^-\), or (ii) are derived from other xor-clauses using one of Inference-rules.

Conflict sets can be classified into three types according to the maximum number of variables in the clauses. A conflict set is simple if it has only xor-clauses with at most one variable, regular if the maximum number of variables is two and complex if it is not simple nor regular.
Definition 6. Let $\Sigma_{conf}$ be a conflict set. The conflict clause of the conflict set $\Sigma_{conf}$ is an or-clause $C_{conf}$ which is logically equivalent to $\neg \Sigma_{conf}$.

When considering the models for a set of xor-clauses, the set of xor-clauses is interpreted as a conjunction. A simple conflict set contains only xor-clauses with one variable, so it can be straightforwardly converted to a conjunction of literals which in turn can be negated to produce a conflict clause that is logically equivalent to the conflict set. By Lemma 3 only simple conflict sets are guaranteed to have conflict clauses, so in the remainder of the paper we consider only simple conflict sets. Exploitable of other types of conflict sets is left for future work.

The purpose of conflict clause is to prune the search space of a SAT solver operating the XOR module. A conflict clause is supplied as the return value of the method explain in Table 1. It conveys the necessary bits of information from the xor part of the problem instance to the SAT solver so that the SAT solver can justifiably undo one or more of its decisions. The SAT solver may store the information to avoid redoing the same decisions in further search. That is why it is important that a conflict clause does not eliminate any models of the problem instance. We will prove now that a conflict clause is a logical consequence of the problem instance.

Theorem 2. Let $D = C_1, \ldots, \perp$ be a xor-refutation, $G = \langle V, E, r, d \rangle$ the implication graph of $D$. $\langle V_a, V_b \rangle$ a cut of the implication graph $G$, $\Sigma_{conf}$ the conflict set of the cut $\langle V_a, V_b \rangle$, $\Sigma_{app}$ the supporting set of the cut $\langle V_a, V_b \rangle$, and $C_{conf}$ the conflict clause of the conflict set $\Sigma_{conf}$. It holds that $\Sigma_{app} \models C_{conf}$.

Proof. As the xor-clause $\perp$ can be derived from the set of xor-clauses $(\Sigma_{app} \cup \Sigma_{conf})$ using Inference-rules which are sound (Theorem 1), it holds that $(\Sigma_{app} \cup \Sigma_{conf}) \models \perp$. This implies that the formula $\neg (\Sigma_{app} \cap \Sigma_{conf})$ is valid. The formula can be converted into an implication: $\neg (\Sigma_{app} \cap \Sigma_{conf}) \iff \neg \Sigma_{app} \lor \neg \Sigma_{conf} \iff \Sigma_{app} \rightarrow \neg \Sigma_{conf}$. By definition the conflict clause $C_{conf}$ is logically equivalent to $\neg \Sigma_{conf}$ so $\neg \Sigma_{conf}$ be substituted with $C_{conf}$ which results in a valid formula $\Sigma_{app} \rightarrow C_{conf}$ and concludes the proof. □

When a conflict is derived, it is intuitive to think that the last decided truth value was wrong (with regard to the other decisions on truth values of the variables). The nodes of the last decision level are logical consequences of the last decided truth value. That is why we include exactly one node of the last decision level in the explanation for a conflict. In addition to the node that corresponds to the last decided truth value, there may be other nodes of the last decision level that are good candidates for the explanation. These are called unique implication points.

Definition 7. (unique implication point). Let $D = C_1, \ldots, \top$ be a xor-refutation and $C_{decide}$ in $D$ be a xor-clause introduced using Decide$^+$ or Decide$^-$. Let $G = \langle V, E, r, d \rangle$ be the implication graph of $D$. Let $V_{decide} \in V$ be the node corresponding to $C_{decide}$. If there is a path from $V_{decide}$ to a conflict node that does not include any nodes whose rule label is Decide$^+$ or Decide$^-$, each node that belongs to all paths from $V_{decide}$ to the conflict node is a unique implication point (UIP).

Each unique implication point defines a number to cuts of the implication graph. Experimental results suggest that a unique implication point (there can be several) that has the shortest path to a conflict node (UIP scheme) yields the most useful explanations on average. However, in order to efficiently convert the conflict set to a CNF formula, we consider only simple conflict sets.

Definition 8. (conflict set of UIP). Let $G = \langle V, E, r, d \rangle$ be an implication graph, $s \in V$ be a unique implication point and $\text{cuts}(s)$ denote the subset of the cuts of $G$ such that for each cut in $\text{cuts}(s)$ it holds that the UIP $s$ is in the first part of the cut and has a direct successor node in the second part of the cut. The conflict sets of the UIP $s$ are the conflict sets of the cuts $\text{cuts}(s)$.

Example 6. Three of the cuts of the implication graph in Figure 7 have been marked. The nodes 8 and 9 are unique implication points of the node 8. The cut 1 defines the conflict set $\{x_1 \oplus x_2 \oplus \top, x_3 \oplus x_4 \}$. The cut 2 defines the conflict set $\{x_1, x_2 \oplus x_3 \oplus \top, x_5 \}$ which is a conflict set of the UIP node 9. The cut 3 defines the conflict set $\{x_1, x_2 \}$ which is a conflict set of the UIP node 8. The conflict clause of the simple conflict set $\{x_1, x_2 \}$ is $\neg x_1 \lor \neg x_2$.

The next theorem states that a conflicting xor-derivation always has a conflict set and that there is a conflict set which contains only xor-clauses introduced using Decide$^+$- or Decide$^-$-rules.
Theorem 3. Let \( D = C_1, \ldots, \bot \) be a xor-refutation, \( \Sigma_{\text{premises}} \subseteq D \) be the set of xor-clauses introduced using Introduce-rule, \( \Sigma_{\text{decide}} \subseteq D \) be the set of xor-clauses introduced using Decide\(^{-}\)- or Decide\(^{+}\)-rules, \( \Sigma_{\text{derived}} \subseteq D \) be the set of xor-clauses derived using Inference-rules, and \( G = (V, E, r, d) \) be an implication graph of \( D \). There is a cut of \( G \) such that the corresponding conflict set \( \Sigma_{\text{conflict}} \subseteq \Sigma_{\text{decide}} \).

Proof. There is a cut \((V_a, V_b)\) of \( G \) such that \((\Sigma_{\text{premises}} \cup \Sigma_{\text{decide}}) \subseteq V_a\) because for all nodes \( t \in V \) it holds that if \( t \in \Sigma_{\text{premises}} \) or \( t \in \Sigma_{\text{decide}} \), then there are no incoming edges to \( t \). All nodes in \( \Sigma_{\text{derived}} \) have incoming edges so they can be in the second part \( V_b \) of the cut. Therefore, it is always possible to define a cut such that \( V_a = \Sigma_{\text{premises}} \cup \Sigma_{\text{decide}} \) and \( V_b = \Sigma_{\text{derived}} \). By definition, a conflict set never contains any xor-clauses in \( \Sigma_{\text{premises}} \) and all xor-clauses of a conflict set are selected from the first part \( V_a \) of a cut. It follows that \( \Sigma_{\text{conflict}} \subseteq \Sigma_{\text{decide}} \). \( \square \)

8 Implementation

The proof-of-concept implementation, LibXorSat hereafter, implements the interface described in Table 1 in page 5.

As a set of xor-clauses corresponds to a system of linear equations which can be represented as an augmented matrix, a convenient way of modelling it as a data structure would be a \( c \times n \) bit-array \( A \) where \( c \) is the number of clauses and \( n \) is the number of variables. The Boolean value of the element \( A[i, j] \) would indicate whether the xor-clause \( C_j \) contains an occurrence of the variable \( x_j \). However, the linear part of the problems we are interested in - for instance, cryptanalysis of stream ciphers - tends to be sparse when represented as a matrix. If a bit-array representation were to be used, much of the array would remain unused yet still occupying memory. Also, Gaussian elimination is performed only at the end of the normal use case scenario of LibXorSat so the xor-clauses cannot grow in length. That is why occurrences of variables are represented as lists excluding zero coefficients. This results in a memory usage of the order \( O(c) \) instead of \( O(c \times n) \).

A high level illustration of the design is shown in Figure 5 as UML class diagram [14]. Only the most relevant operations and attributes are listed. The words in Italic refer to class names in the UML class diagram in this section. The top-level component of LibXorSat is Solver which operates all the other classes. Its methods include the operations listed in Table 1.

Before any xor-clause can be added, the variables that are to be used in the xor-clauses have to be declared (the method add-variable). This allocates a data structure (Variable) that will hold references to xor-clauses that will have occurrences (Instance) of the added variable. When a xor-clause is added (the method add-clause), a data structure (Clause) is allocated and two-way links between variables of the xor-clause and the xor-clause are stored. Two-way referencing is needed to support addition and removal of variable occurrences efficiently.

Typical usage of LibXorSat continues by assigning values to variables (the method assign) and then asserting whether the set of xor-clauses is still satisfiable. A variable is assigned by eliminating its occurrences from corresponding xor-clauses. The xor-clauses are represented in normal form so if the truth value of the variable is true, then the flag indicating the presence of the symbol \( \top \) is flipped in each of the xor-clauses that had an occurrence of the variable. If a xor-clause has one or two remaining occurrences of other variables after the assignment, one of Inference-rules is applied. This may result in other variables being assigned and more applications of the inference rules. The process is called propagation (see Algorithm 11) and it is continued until all xor-clauses with one or two variable occurrences have been tested for possible rule applications.

If an assignment (a value assigned to a variable) leads to a conflict, that is, the initial set of xor-clauses augmented with the unary xor-clauses of the assignments is no longer satisfiable, a conflict clause is calculated which can then be retrieved by invoking the method explain. The conflict clause is derived by undoing the operations and building a conflict set. An initial conflict set is prepared by adding the conflicting node in it. Then the operations are undone and the contents of the conflict set are expanded until the conflict set (i) is simple, (ii) has at most one node on the last decision level, and (iii) does not consist of two xor-clauses \( A \) and \( \neg A \) where \( A \) is a variable. The idea of the conflict analysis is illustrated in Algorithm 12.

A SAT solver proves the satisfiability or the insatisfiability of a CNF formula by assigning values to variables and then backtracking when a satisfying truth assignment is not found in the current search branch. In order to support backtracking search, the state of LibXorSat can be stored by setting a backjump point (BackjumpPoint, the method set-backjump). All modifications to the data structures caused by later assignments will be recorded. One of the previously recorded states can be restored by backjumping to a backjump point.
(the method backjump). The operations on xor-clauses are destructive, that is, the data structures are modified in place. Modifications caused by primitive operations (Assign, Substitute) are stored in a data structure (OperationStack) so that they can be undone and redone during backtracking and conflict analysis. A variable occurrence (Instance) is in fact hidden instead of removing it when a variable is assigned so that it can be restored later in constant time. The substitute operation (Substitute) may add new occurrences of variables. Such an occurrence is marked as temporary in order to remove it when undoing the operation instead of hiding it.

After the SAT solver has assigned the variables of the CNF formulas, satisfiability of the simplified xor-clauses is checked by performing Gaussian elimination (Gauss, the method solve). Gaussian elimination does destructive modifications to the data structures which would be computationally expensive to undo so a copy of the xor-clauses is operated instead. The remaining unknown variables are assigned after Gaussian elimination and a model is extracted.

As the operations are destructive, the data structure for a xor-clause (Clause) corresponds to possibly several nodes in the implication graph. The active data structures of the xor-clauses correspond to the leaf nodes of the implication graph. In order to save memory, only the information needed to define the structure of the implication graph is recorded. Each modification to a xor-clause produces a new xor-clause which is labelled with a unique identifier. The data structure for xor-clauses stores a number of records (IdInfo) which capture the identifiers, the identifiers of the parents and the decision level of each xor-clause (node) the data structure represents.

Algorithm 1 Propagate(clause)
1: if clause is marked as propagated then
2: return
3: end if
4: MARK-AS-PROPAGATED (clause)
5: N ← NUM-FREE-VARIABLES (clause)
6: if N = 1 then
7: V ← VARIABLES-OF(clause)
8: T ← has-top(clause) = false
9: O ← AssignOp(clause, V, T)
10: push-op(O)
11: do-operation(O)
12: else if N = 2 then
13: Va, Vb ← VARIABLES-OF(clause)
14: T ← has-top(clause) = false
15: O ← SubstituteOp(clause, Va, Vb, T)
16: push-op(O)
17: do-operation(O)
18: end if

9 Experimental Results

We evaluated the scalability of our approach on randomized test instances based on 3-regular bipartite graphs [9]. The DPLL-style use of the module was simulated by first adding xor-clauses and then assigning random truth values for randomly picked unassigned variables until a conflict or a solution was found. When a conflict or a solution was found, a random number of the assignments was canceled by performing a backjump. A new test instance was generated after 50 backjumps. The test was run 10 minutes for each test setup (number of variables n = 8, 16, . . . , with/without conflict analysis).

We measured the median number of truth value assignments per second, the median number of propagation steps per assignment and the maximum memory usage. The tests were run on an Intel Core 2 Duo 2.33GHz with 4096 KB of L2 cache memory and on an Intel Core 2 Duo 3.00GHz with 6144 KB of L2 cache memory. The test results are shown in Figures [9] and [10]. The vertical bars at each point of the curves illustrate the standard deviation.
In Figure 9 there are four curves illustrating the “external” performance of the module, i.e. how many invocations of the method assign per second the module can execute. The test was run on both computers with (analyzing) and without (no analyzing) conflict analysis. Considering the test instances as $n \times n$ matrices where $n$ is the number of variables, the matrices with 8 and 16 variables are so dense that on average only a small number of propagation steps per assigned variable is required to derive a conflict which (see Figure 10) explains why the external performance peaks with small test instances. As the size of the matrices increases,
Algorithm 2: AnalyzeConflict(clause)

1: ADD-TO-SET(set, clause)
2: while (IS-TRIVIAL(set)
   or not IS-SIMPLE(set)
   or NODES-AT-THE-LAST-LEVEL(set) ≠ 1) do
3:   modifiedClauses ← UNDO-OPERATION()
4:   for all clause ∈ modifiedClauses do
5:     numLast ← NODES-AT-THE-LAST-LEVEL(set)
6:     if (clause ∈ set)
       and (NUM-VARIABLES(clause) > 1
       or (not IS-LAST-LEVEL(clause) and IS-TRIVIAL(set))
       or (IS-LAST-LEVEL(clause) and numLast = 1)) then
7:       REMOVE-FROM-SET(set, clause)
8:     left, right ← PARENTS(clause)
9:     ADD-TO-SET(set, left)
10:    ADD-TO-SET(set, right)
11:   end if
12: end for
13: end while

the length of propagation chains (number of propagations per external truth assignment) increases as well. This results in lower external performance. Eventually the matrices become so sparse that propagation chains begin to shorten. This explains the increase in the external performance on test instances with 128 to 4096 variables. As the test instances grow further, they no longer fit in the cache memory which results in a sudden drop of performance because the implementation does not use cache locality techniques. On the computer with 6144 KB of cache memory, the performance drops later than on the computer with the smaller cache. After the cache memory runs out, the external performance degrades gradually as the size of test instances increases. In this test, the memory usage of the module is linear with respect to number of variables with less than 1kB of memory per variable.

10 Conclusions and Future Work

We have presented a decision procedure and a proof-of-concept implementation for deciding the satisfiability of xor-clauses. We have described a method for conflict analysis that produces conflict explanations that are likely to prune a part of the search space through DPLL clause learning.

The preliminary results suggest that our approach is scalable enough to handle instances with hundreds of thousands of variable occurrences. We are currently working on integrating the LibXorSat module into the SAT solver MiniSat in order to compare the combined approach with an unmodified version of the solver. We believe that representing and operating xor-clauses in a compact form instead of converting them to CNF formulas and benefiting from the conflict-learning capabilities of current SAT solvers is a viable approach for increasing the performance of SAT solvers when solving problems with equivalences/xors.

The conflict analysis method we presented produces simple conflict sets, that is, explanations that can be easily converted to CNF formulas. This is done to facilitate the integration to a SAT solver. However, we suspect that using more complex explanations and more sophisticated translations of conflict sets can prune even a larger part of search space. We will continue investigating possibilities of conflict-driven learning also in the XOR module in such a way that the part of an explanation that does not translate well to CNF would be added in the XOR module as a learned xor-clause and the rest would be converted to CNF.

Also, besides producing explanations for conflicts, the XOR module can be used to derive truth values for variables by the use of inference rules. These “implied literals” can be propagated back to the SAT solver along with an explanation similar to an explanation for a conflict. The explanation for an implied literal can be derived using the same algorithm used in conflict analysis. This kind of two-way propagation might be useful for solving disjoint clusters of xor-clauses.

Representing equivalences separately from the CNF formula is likely to be beneficial but the translation that splits a problem to CNF- and XOR-parts can lose important information about the structure of the problem.
which could be exploited during the search. Boolean circuits [13] preserve all the relationships between variables in a compact form. We plan to study the use of Boolean circuits in such a way that the original problem description is split to CNF- and XOR-parts but also kept as a Boolean circuit which is consulted during the search for additional implied equivalences.

As a case study, we are interested in testing the applicability of the combined approach (a SAT solver with LibXorSat) as a component of tools for logical cryptanalysis. A typical stream cipher encrypts the plaintext by XOR operations with a pseudo-random bit stream [10]. XOR expressions form chains where the output of one operation is used as an input of the next operation. Also, XOR expressions are short involving only a small non-fixed number of variables. The inferences rules are likely to work well with this kind of instances.

The experimental results of the study suggest that the module can be integrated into a SAT solver and that it is likely to enhance the search. The actual integration is left to future work.

Acknowledgements

I wish to thank Prof. Ilkka Niemelä and Docent Tommi Junttila for their valuable comments. The financial support of the Academy of Finland (project 122399) is gratefully acknowledged.

References


Figure 10: Propagation rate


[16] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In SAT [I], pages 244–257.

