TKK Reports in Information and Computer Science
Espoo 2010 TKK-ICS-R32

EXTENDING SAT SOLVER WITH PARITY CONSTRAINTS

Tero Laitinen

TKK Reports in Information and Computer Science
Espoo 2010 TKK-ICS-R32

EXTENDING SAT SOLVER WITH PARITY CONSTRAINTS

Tero Laitinen

Aalto University School of Science and Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science

Aalto-yliopiston teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietojenkäsittelytieteen laitos

Distribution:
Aalto University School of Science and Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science
PO Box 15400
FI-00076 AALTO
FINLAND
URL: http://ics.tkk.fi
Tel. +358 9 470 01
Fax +358 9 470 23369
E-mail: series@ics.tkk.fi

ISBN 978-952-60-3223-8 (Print)
ISBN 978-952-60-3224-5 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)
URL: http://lib.tkk.fi/Reports/2010/isbn9789526032245.pdf

AALTO ICS
Espoo 2010

ABSTRACT: Current methods for solving Boolean satisfiability problem (SAT) are scalable enough to solve discrete nonlinear problems involving hundreds of thousands of variables. However, modern SAT solvers scale poorly with problems involving parity constraints (linear equations modulo 2). Gaussian elimination can be used to solve a system of linear equation effectively but it cannot be applied as such when the problem description also contains nonlinear constraints. A SAT solver typically reads the problem description in conjunctive normal form (CNF). Representing parity constraints in CNF results in an inefficient encoding which partly makes solving the problem harder. Also, the deduction methods used in SAT solvers are not efficient when solving problems involving parity constraints.

This report develops an efficient xor-reasoning module for deciding the satisfiability of a conjunction of parity constraints and studies alternative ways to integrate the xor-reasoning module into a modern conflict-driven clause learning SAT solver. The novelty of the approach is the combination of conflict-driven clause learning techniques (CDCL) with equivalence reasoning enhancing deduction capabilities of CDCL SAT solvers beyond unit propagation on the CNF formula. The presented proof system and the abstract model for computing clausal explanations for inconsistent valuations of variables allow for different possible implementations. Key design principles along with alternative ways to compute clausal explanations and to integrate the xor-reasoning module into a SAT solver are presented, analyzed, and compared. We have integrated the xor-reasoning module into a state-of-the-art CDCL SAT solver, minisat. The applicability of the hybrid approach is evaluated experimentally and compared with a number of modern SAT solvers on three challenging benchmarks: randomly generated regular linear problems, a known keystream attack on the stream cipher Trivium, and a known plaintext attack on the block cipher DES. The results are promising: the number of heuristics decisions needed typically decreases without causing unbearable computational overhead. Larger problem instances may even be solved faster by minisat with the xor-reasoning module than by the unmodified version.

KEYWORDS: SAT, Boolean logic, parity constraint, conflict-driven

CONTENTS

1 Introduction 7
- 1.1 Related Work . . . 10
- 1.2 Comparison with Existing Work . . . 11
- 1.3 Organization of the Report . . . 12

2 Preliminaries 12
- 2.1 Basic Definitions . . . 12
- 2.2 Overview of Solving CNF-XOR SAT Problem with SAT Solver 14
- 2.3 Combined Approach for Solving CNF-XOR SAT Problem . . 15

3 XOR-reasoning module 16
- 3.1 Inference Rules . . . 17
- 3.2 Defining Reason Set for Logical Consequence . . . 21
- 3.3 Computing Reason Set for Logical Consequence . . . 26
- 3.4 Prioritizing Inference Rules . . . 30
- 3.5 XOR-Implied Literals . . . 30
- 3.6 XOR-Internal Variables . . . 33
- 3.7 Redundancy in Reason Set . . . 36

4 CNF/XOR Integration 40
- 4.1 CDCL SAT Solver . . . 41
- 4.2 CDCL/XOR SAT Solver . . . 43
- 4.3 Fully Saturated XOR-Propagation . . . 45
- 4.4 Minimal XOR-Propagation . . . 47
- 4.5 Postponed XOR-Propagation . . . 48
- 4.6 Handling XOR-Implied Literals in SAT Solver . . . 54

5 Experimental Results 56
- 5.1 Block Cipher DES . . . 57
- 5.2 Randomly Generated Linear Problems . . . 59
- 5.3 Stream cipher Trivium . . . 62

6 Conclusions 68

References 70

1 INTRODUCTION

The propositional satisfiability problem (SAT) is to decide whether the variables of a propositional (Boolean) formula can be assigned in such a way that the formula evaluates to "true". Given a propositional formula, a SAT solver is a computer program that finds a valuation for the variables of the formula making the formula evaluate to "true" or proves that none exists. SAT solvers have been successfully applied to many problem domains including AI planning, model checking of software systems, and package management in software distributions [25]. A description of the history of the propositional satisfiability problem along with techniques for solving SAT problems is presented in [8].

Propositional logic is a natural formal language for expressing computation doable by a computer. It consists of two-valued variables and Boolean formulas that evaluate either to "true" or "false" depending on the valuation of the variables occurring in the formula. There is a close connection between propositional logic and computer hardware. Boolean circuits can be seen as abstract representation of computer circuits. Computer circuits consist of logic gates which are physical implementations of logical connectives such as AND and OR. From this point of view, it is intuitively reasonable that any fixed-length computation doable by a computer can be represented as a Boolean circuit and then converted to a Boolean formula. The generality of the SAT problem has a price, though. Results from computational complexity theory suggest that it is likely that in the worst case, solving an instance of the SAT problem takes an exponential number of computational steps with respect to the number of variables occurring in the Boolean formula [30]. However, real-world problems tend to have structure that can be exploited by a SAT solver. Modern SAT solvers scale up to problem instances involving hundreds of thousands of propositional variables. Most modern SAT solvers such as `minisat` [12], `picosat` [7], and `Chaff` [27] are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [11] which is an efficient decision procedure to solve the satisfiability of a Boolean formula expressed in conjunctive normal form (CNF). Research on solving the SAT problem has been active and there has been a number of important improvements since the 1990s which have had a drastic effect on the performance of the SAT solvers. One of the most important techniques is called conflict-driven clause learning [32, 38]. The idea of conflict-driven clause learning is to store information from the explored search space in order to prune the part of the remaining unexplored search space that is known not to contain any solutions to the problem.

Use of the DPLL algorithm requires that a SAT problem is expressed in conjunctive normal form in which few problems have natural representations. Thus, SAT problems are usually converted to CNF, e.g. using the translation by Tseitin [37]. How a SAT problem is encoded in CNF has a significant effect on how computationally demanding it is to solve the problem, so research on efficient CNF conversions has been active (see e.g. [18, 3, 23]).

Despite the promising improvements in SAT solving techniques, there are problems for which it is difficult to define an encoding in conjunctive

normal form that would not suffer from the exponential worst case solving time. Experimental results suggest that large-scale use of certain logical connectives, such as the logical equivalence $\leftrightarrow$ or its negative counterpart, the exclusive or (xor), results in problem descriptions in CNF that modern SAT solvers scale poorly with [13]. A set of parity constraints (a conjunction of xor-clauses) effectively corresponds to a system of linear equations modulo 2. Therefore, if a problem instance consists only of parity constraints, it can be efficiently solved using Gaussian elimination [36], which is a polynomial-time algorithm for solving systems of linear equations. However, Gaussian elimination cannot be applied if nonlinear connectives are also used in the problem description. In instances of problem domains such as logical cryptanalysis [26] and circuit verification [31], a substantial amount of constraints consists of xor-clauses and in part because of that larger problem instances from these domains are intractable for current SAT solvers.

As SAT solvers scale poorly with parity constraints, an intuitive way to address the limitations of CNF-based reasoning is to separate parity constraints from the CNF part, leading to the *cnf-xor SAT problem* which is to decide whether the variables of a conjunction of disjunctions (or-clauses) and xor-clauses can be assigned in such a way that it evaluates to "true". By separating parity constraints from the CNF part, a separate solver module can be used to reason about parity constraints in the most effective way. The solver module can then be integrated to many modern SAT solvers. By using this kind of modular approach, the cutting edge of SAT solving technology can be continuously exploited. A state-of-the-art approach for integrating such a solver module into a conflict-driven clause learning SAT solver is the DPLL(T) framework by Nieuwenhuis, Oliveras, and Tinelli [28], which was developed as a general method to solve an extension of the SAT problem, so-called Satisfiability Modulo Theories [5].

Research Problem This work studies how the cnf-xor SAT-problem can be solved using the DPLL(T) framework enabling to integrate a cutting edge (CDCL-based) SAT solver and a special xor-reasoning module handling the xor part.

The approach boils down to two key research questions: what is a suitable xor-reasoning module for this framework and how to integrate it to a CDCL-based SAT solver in an effective way. The use of a CDCL-based SAT solver as a basis leads to the following specific requirements for the xor-reasoning module:

- **Incremental operations** : The SAT solver supplies deduced truth values one by one and expects intermediate results after each propagated truth value. As the state of the SAT solver is modified incrementally, it is important that any results produced by the xor-reasoning module do not have to be computed from the initial state each time when the state of the SAT solver is modified.

- **Backtrackable state** : DPLL-based algorithms are based on backtracking search. It must be possible to restore the state of the xor-reasoning module to any consistent preceding state without excess computational overhead.

- **Strong proof system** : Use of the xor-reasoning should make the proofs of the SAT solver significantly shorter in order to reduce the overall solving time. The xor-reasoning module has to be able to infer results that cannot be deduced by unit propagation in CNF alone.

- **Explainable deductions** : A conflict-driven clause learning SAT solver needs to be able to explain how a truth value of a variable was deduced when a conflict, an inconsistent valuation for variables, occurs.

- **Minimal interface** : Integrating the xor-reasoning module to a SAT solver capable of conflict-driven clause learning should be a fairly reasonable effort.

- **Efficiently implementable** : Data structures and algorithms used in SAT solvers have been perfected by the research community for decades. They are extremely well fine-tuned for speed, capable of performing even millions of deduction steps in a second, so in order to benefit from the xor-reasoning module also with respect to the overall solving time, the implementation of the xor-reasoning module has to be fast.

This report develops the xor-reasoning module according to the specified requirements comparing and analyzing various design choices and alternative strategies for integrating the xor-reasoning module to a SAT solver capable of conflict-driven clause learning. The main contributions of the work are:

- **An efficient DPLL(T) solver module for parity constraints** : We have implemented a proof-of-concept prototype of the xor-reasoning module. The design of the xor-reasoning module is a careful compromise between the requirements, enhancing the search by shortening proofs without adding an unbearable computational overhead. The most important contribution of this report is the combination of conflict-driven clause learning and equivalence reasoning. The presented proof system and abstract model for defining clausal explanations for inferred truth values allow for many different implementations. We present alternative methods for computing clausal explanations. Also, the treatment of variables that have occurrences only in xor-clauses is an interesting topic having a significant effect on the lengths of proofs.

- **Strategies for integrating the xor-reasoning module to a conflict-driven clause learning SAT solver** : We have integrated the proof-of-concept implementation of the xor-reasoning module to a state-of-the-art conflict-driven clause learning SAT solver minisat, resulting in a hybrid solver xor-minisat. There are many ways for using the xor-reasoning module as a subroutine of the SAT solver, ranging from doing as much reasoning in the xor-reasoning module as possible to postponing the use of xor-reasoning module as long as possible. In order to examine alternative integration strategies, an abstract model of the SAT solver is used to analyze the proposed strategies in detail. Also, alternative ways for using clausal explanations provided by the xor-reasoning module in the SAT solver are studied.

The integration strategy of the xor-reasoning module to a SAT solver follows the DPLL(T) framework with one addition. In the context of the DPLL(T) framework, the use of the xor-reasoning module is considered *theory propagation*. When the degree of eagerness of theory propagation is low, that is, the use of xor-reasoning module is postponed as long as possible, in addition to the normal strategy of computing an explanation for a conflict that occurs during theory propagation, it is possible to retrospectively apply the results of the deduction done by the xor-reasoning module in order to shorten the remaining search. This procedure is described in detail in Section 4.5. Although we define the procedure only in the context of parity constraints, it can be generalized to any SAT solver-based SMT solver.

- **Experimental evaluation of the proposed approach** In order to illustrate how well our approach scales in practice, we evaluate and compare our solver xor-minisat experimentally on three challenging benchmarks : known-plaintext 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. In addition to comparing how the performance of the solver minisat changes when the xor-reasoning module is used, a number of solvers that employ techniques for dealing with parity constraints is also included in the comparison.

1.1 Related Work

Intractability of problems involving a substantial amount of parity constraints for SAT solvers is well acknowledged and considerable research has been directed towards enhancing SAT solvers with xor (equivalence) reasoning techniques [15]. In this section, we describe some approaches to solving the research problem of the report proposed by others.

Baumgartner and Massacci [6] develop a decision procedure for conjunctions of or-clauses and xor-clauses based on the original DPLL procedure without conflict-driven clause learning.

The solver EqSatz by Li [22] 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. It does not employ conflict-driven clause learning.

The solver march_eq by Heule and van Maaren [14] extracts equivalences from a CNF formula and produces a minimal solution for the linear part before starting the actual DPLL-based search procedure by eliminating heuristically picked dependent variables in the conjunction of equivalences so that all satisfiable truth assignments can be defined in terms of the remaining independent variables. 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 [10] is a hybrid SAT solver in the sense that

it employs both look-ahead and conflict-driven clause learning techniques. The solver moRsat extracts xor-clauses from CNF and performs some preprocessing on them. In moRsat, xor-clauses are stored as normal disjunctions but the watched literal scheme presented in [27] requires a small modification in order to support unit propagation effectively on xor-clauses: the negations of the two watched literals are tracked as well. When a xor-clause becomes implying, the literals in the clause are negated in such a way that the implied literal is true and each literal in the implying part is false.

The solver cryptominisat by Soos et al. [33] accepts a mixture of CNF and xor-clauses as its input. The xor-clauses are like ordinary disjunctions in the solver's data structures but change appearance according to the assigned literals as in moRsat. 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.

The solver 2clseq by Bacchus [4] enhances search by performing three types of reasoning in addition to regular unit propagation: i) in *binary resolution* all possible pairs of binary clauses are tested in a resolution step in order to yield more binary clauses ii) a technique called *hyper-resolution* is used to shorten n-ary clauses by successive applications of binary resolution in order to yield more binary clauses, and iii) *equality reduction* finds pairs of equivalent literals and removes occurrences of the other literal.

The solver lsat by Ostrowski et al. [29]. performs preprocessing that reconstructs structural information (including equivalences) from the CNF formula which is exploited during the search.

1.2 Comparison with Existing Work

In this section, we point out the key differences between our work and approaches relevant to the research problem proposed by others. We compare the solvers presented in the previous section with respect to how xor-clauses can be encoded as an acceptable input for the solvers, whether the problem description is preprocessed before starting the actual search procedure, the nature of the search procedure, and the use of equivalence reasoning techniques.

Input Format. The solvers EqSatz, moRsat, march_eq, 2clseq, and lsat read the problem description in DIMACS CNF and extract parity constraints from CNF. Like in cryptominisat, instead of pattern matching parity constraints from CNF where parity constraints suffer from an unoptimal encoding, our solver xor-minisat accepts the linear part of the problem description is presented as-is.

Preprocessing. Our solver xor-minisat does not perform any kind of preprocessing on the problem instance. The solvers march_eq, 2clseq and lsat perform extensive preprocessing in order to simplify the problem and to extract additional information that can be exploited during the search. The solver moRsat performs some preprocessing on parity constraints, too.

Look-ahead vs Conflict-driven. Look-ahead SAT solvers have advanced heuristics for determining a good unassigned variable to be assigned when truth values for variables cannot be deduced. A detailed description of look-ahead techniques is presented in [16]. The solvers EqSatz, lsat, 2clseq, and march_eq are purely look-ahead-based solvers. Like our solver xor-minisat, the solver cryptominisat is purely conflict-driven. The solver moRsat performs look-ahead branching heuristics in addition to conflict-driven clause learning.

Equivalence Reasoning. Like our solver xor-minisat, the solvers EqSatz, march_eq, 2clseq, and lsat have separate data structures and deduction rules for keeping track of (in)equivalences between variables. As EqSatz, our solver xor-minisat uses binary equivalences/xors to simplify clauses by substituting variable occurrences. The solvers moRsat and cryptominisat do not perform equivalence reasoning using binary xor-clauses.

1.3 Organization of the Report

The report is organized as follows. First, basic notation and definitions along with an overview of the approach are introduced in Chapter 2. Chapter 3 describes the proof system and relevant design principles of the xor-reasoning module. Chapter 4 develops an abstract model of a SAT solver enhanced with xor reasoning in order to explore possible alternatives for integrating the xor-reasoning module into a SAT solver. Chapter 5 presents an experimental evaluation of the approach and Chapter 6 concludes the report.

2 PRELIMINARIES

This chapter establishes notation and basic definitions used in the rest of the report, and gives an overview of the problem setting.

2.1 Basic Definitions

In this section we introduce notation and definitions that are used to describe the proof system of the xor-reasoning module and the formal model of integrating the xor-reasoning module to a SAT solver.

An *atom* is either a propositional variable or the symbol $\top$ which denotes the truth value "true". A *literal* is an atom A (*positive literal*) or its negation $\neg A$ (*negative literal*). We use the symbol $\bot$ as a shorthand to denote the literal $\neg\top$. An *or-clause* is an expression $C = L_1 \vee \cdots \vee L_n$ where $L_1, \ldots, L_n$ are literals and the symbol $\vee$ stands for the non-exclusive logical OR. The empty or-clause is denoted by $\bot$. A *conjunction* of or-clauses is an expression $\phi_{or} = C_1 \wedge \cdots \wedge C_n$ where $C_1, \ldots, C_n$ are or-clauses and the symbol $\wedge$ stands for the logical AND. A conjunction of or-clauses is also called a formula in *conjunctive normal form* or a *cnf-formula*. A *xor-clause* is an expression $X = L_1 \oplus \cdots \oplus L_n$ where $L_1, \ldots, L_n$ are literals and the symbol $\oplus$ stands for the exclusive logical OR. Empty xor-clause is denoted by $\bot$. A conjunction of xor-clauses is an expression $\phi_{xor} = X_1 \wedge \cdots \wedge X_n$ where $X_1, \ldots, X_n$ are

a	b	c	$\neg a \vee b$	$b \vee \neg c$	$a \oplus c$	$b \oplus c$	ϕ
0	0	0	1	1	0	0	0
0	0	1	1	0	1	1	0
0	1	0	1	1	0	1	0
0	1	1	1	1	1	0	0
1	0	0	0	1	1	0	0
1	0	1	0	0	0	1	0
1	1	0	1	1	1	1	1
1	1	1	1	1	0	0	0

Figure 1: Truth table illustrating satisfiability of a cnf-xor-formula where "0" stands for the truth value "false" and "1" denotes the truth value "true"

xor-clauses. A *clause* is either an or-clause or a xor-clause. A *cnf-xor-formula* $\phi = \phi_{\text{or}} \wedge \phi_{\text{xor}}$ is a conjunction where ϕ_{or} is a conjunction of or-clauses and ϕ_{xor} is a conjunction of xor-clauses.

A (complete) *truth assignment* M is a set of atoms containing at least the atom $\top$. We define that $M \models A$ iff $A \in M$, where A is a positive literal. The operator $\models$ is read as "is a model for" or "satisfies". For negative literals of type $\neg A$ we define that $M \models \neg A$ iff $M \models A$ does not hold. A literal is *satisfied* by M iff $M \models L$. An or-clause $C = L_1 \vee \cdots \vee L_n$ is satisfied by M ($M \models L_1 \vee \cdots \vee L_n$) iff at least one of the literals in C is satisfied by M. A cnf-formula $\phi = C_1 \wedge \cdots \wedge C_n$ is satisfied by M ($M \models C_1 \wedge \cdots \wedge C_n$) iff all or-clauses of ϕ are satisfied by M. A xor-clause $X = L_1 \oplus \cdots \oplus L_n$ is satisfied by M ($M \models L_1 \oplus \cdots \oplus L_n$) iff an odd number of literals in X are satisfied by M. A cnf-xor-formula $\phi = \phi_{\text{or}} \wedge \phi_{\text{xor}}$ is satisfied by M ($M \models \phi_{\text{or}} \wedge \phi_{\text{xor}}$) iff ϕ_{or} and ϕ_{xor} are satisfied by M. The negated operator $\not\models$ is read as "is not a model for" or "is not satisfied by". For any P for which $M \models P$ is defined, we define that $M \not\models P$ holds iff $M \models P$ does not hold. Also, P is *satisfiable* iff there is a truth assignment M such that $M \models P$, and *unsatisfiable* when it is not satisfiable.

Definition 1. *Given a cnf-xor-formula ϕ, the cnf-xor SAT problem is to decide whether there is a truth assignment M such that $M \models \phi$.*

Example 1. *The expression $\phi = (\neg a \vee b) \wedge (\neg b \vee \neg c) \wedge (a \oplus c) \wedge (b \oplus c)$ is a cnf-xor-formula involving three variables a, b, and c. The cnf-xor-formula has one model which is the truth assignment $\{ a, b \}$ as shown in Figure 1.*

A *partial truth assignment* $\tilde{M}$ is a set of literals containing at least the literal $\top$. A literal L and its negation $\neg L$ cannot both be in a partial truth assignment. Concerning partial truth assignments, we define that $\tilde{M} \models L$ iff $L \in \tilde{M}$ and $\tilde{M} \not\models L$ iff $\tilde{M} \models \neg L$. Otherwise, the operators $\models$ and $\not\models$ are defined for partial truth assignments as for complete truth assignments. Note that for the partial truth assignment $\tilde{M} = \{ \top \}$ and a literal L (different from $\top$ and $\bot$) neither $\tilde{M} \models L$ nor $\tilde{M} \not\models L$ hold.

Let ϕ_{a} and ϕ_{b} be cnf-xor-formulas. The cnf-xor-formula ϕ_{b} is a *logical consequence* of the cnf-xor-formula ϕ_{a}, denoted by $\phi_{\text{a}} \models \phi_{\text{b}}$, iff for all truth assignments M the following holds : if $M \models \phi_{\text{a}}$, then $M \models \phi_{\text{b}}$.

A *substitution* in a xor-clause X_a is denoted by $X_a\,[A/X_b]$ where A is an atom and X_b is a xor-clause. The substitution $X_a\,[A/X_b]$ defines a new xor-clause identical to X_a except that all occurrences of A are substituted with X_b.

Translation to Normal Form. In the case of xor-clauses, negative literals (except $\bot$) can be eliminated. This simplifies the definition of the rules of the proof system of the xor-reasoning module. Also, a xor-clause may contain redundancy (repeated literals) which can be removed while preserving models of the xor-clause. A xor-clause is in *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 2 for transforming xor-clauses to the normal form are based on the rules presented in [6]. The left hand side is the premise (rule pattern), the right hand side is the conclusion, A is an atom, and C is a xor-clause. The rules are considered applicable for a xor-clause if its literals can be ordered in such a way the xor-clause matches the rule pattern. 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 $\neg a \oplus b \oplus c \oplus c$ is $a \oplus b \oplus \top$, where a, b, and c are propositional variables. In the remainder of the report, we will assume that all xor-clauses of the type $X_a\,[A/X_b]$, where X_a and X_b are xor-clauses and A is an atom, are implicitly transformed to normal form.

Unit propagation. Considering satisfiability of cnf-formulas, an inference method called *unit propagation* can be used to simplify a cnf-formula. As a search done by a SAT solver can be modelled as a series of modifications to a partial truth assignment, we present the unit propagation here with that in mind and leave the cnf-formula unmodified. As an example how unit propagation can be effectively performed, the paper by Zhang and Malik contains a description of effective SAT solving techniques in [39] (unit propagation is referred to as Boolean constraint propagation).

Given a partial truth assignment $\tilde{M}$ and an or-clause $C = L_1 \vee \cdots \vee L_n$ such that $\tilde{M}$ contains the negations of $(n-1)$ of the literals of C and one literal L_n of C (it is assumed here that L_n is the last literal in C) and its negation $\neg L_n$ are not in $\tilde{M}$, the partial truth assignment $\tilde{M}$ can be augmented by the literal L_n because $C \wedge \neg L_1 \wedge \cdots \wedge \neg L_{n-1} \models L_n$. Unit propagation is the deduction step of inferring the augmented partial truth assignment $\tilde{M}_a = \tilde{M} \cup \{\, L_n \,\}$ from the or-clause C and the partial truth assignment $\tilde{M}$. For instance, given the or-clause $C = a \vee b \vee c$ and the partial truth assignment $\tilde{M} = \{\, \neg a, \neg b \,\}$, the partial truth assignment $\tilde{M}_a = \tilde{M} \cup \{\, c \,\} = \{\, \neg a, \neg b, c \,\}$ can be inferred by unit propagation from C and $\tilde{M}$.

2.2 Overview of Solving CNF-XOR SAT Problem with SAT Solver

A problem containing a substantial amount of linear constraints (modulo 2) can be encoded as a cnf-xor-formula $\phi = \phi_{\text{or}} \wedge \phi_{\text{xor}}$ - a conjunction of the nonlinear constraints ϕ_{or} expressed in CNF and the linear constraints ϕ_{xor} expressed as xor-clauses. A solution to the problem is a truth assignment M

$$\begin{aligned} \neg A \oplus C &\rightsquigarrow A \oplus \top \oplus C \\ A \oplus A \oplus C &\rightsquigarrow C \\ A \oplus A &\rightsquigarrow \bot \end{aligned}$$

Figure 2: Rewrite rules for transforming xor-clauses to normal form

that is a model for ϕ.

A typical conflict-driven SAT solver reads the problem description as a cnf-formula ϕ_{or} (e.g. minisat [12]). The SAT solver tries to find a model for the cnf-formula ϕ_{or} (or a proof that none exists) by extending initially empty partial truth assignment $\tilde{M}$ until it includes a literal for each variable occurring in ϕ_{or}. The SAT solver heuristically picks a literal not yet in $\tilde{M}$, adds the picked *decision literal* to $\tilde{M}$, and then checks whether it can infer other literals by unit propagation. If $\tilde{M}$ contains a literal for each variable occurring in ϕ_{or} and $\tilde{M}$ is a model for all the or-clauses in ϕ_{or}, the SAT solver can terminate the search and output the model $\tilde{M}$. The unit propagation may infer a literal whose negation is already in $\tilde{M}$. In this case the SAT solver is in a conflicting state where one of the or-clauses of ϕ_{or} cannot be satisfied by extending $\tilde{M}$. The SAT solver analyzes reasons for the conflict and computes an or-clause C such that $\tilde{M} \models \neg C$ and $\phi_{\text{or}} \models C$. The second condition $\phi_{\text{or}} \models C$ guarantees that the or-clause C does not eliminate any models for ϕ_{or} when C is added to ϕ_{or} in order to prevent the SAT solver from picking the same literals again. The SAT solver then removes literals from $\tilde{M}$ in reverse order until $\tilde{M} \models \neg C$ holds no more. If the cnf-formula ϕ_{or} contains both the unary or-clause (L) and the unary or-clause $(\neg L)$, where L is a literal, then the SAT solver can terminate the search and deem the cnf-formula ϕ_{or} unsatisfiable.

2.3 Combined Approach for Solving CNF-XOR SAT Problem

We plan to extend a conflict-driven SAT solver with a xor-reasoning module in such a way that the cnf-part ϕ_{or} of the cnf-xor-formula $\phi = \phi_{\text{or}} \wedge \phi_{\text{xor}}$ is given to the SAT solver and the xor-part ϕ_{xor} of ϕ is given to the xor-reasoning module. The search is operated by the SAT solver and the xor-reasoning module is used as a subroutine for checking whether the xor-part ϕ_{xor} is still satisfiable with respect to the partial truth assignment being extended by the SAT solver. The SAT solver, thus, needs a method to *assign* a truth value to a variable in the xor-reasoning module. In addition to unit propagation, the SAT solver enhanced with xor-reasoning performs propagation on the xor-part and may extend the partial truth assignment by inference rules on xor-clauses.

If a variable's value can be determined by the SAT solver, the value can be supplied directly to the xor-reasoning module and, thus, avoid performing unnecessary reasoning. The xor-reasoning module applies inference rules (unit propagation and substitution rule based on binary xor-clauses) to in-

fer values for unassigned variables. In order to perform xor-reasoning only when it is likely to be beneficial, the SAT solver is provided with a method to *deduce* a singular logical consequence using the xor-reasoning module.

In case of an inconsistent valuation (conflict), which happens when the SAT solver and the xor-reasoning module disagree on a value of a variable, the assignments that led to the conflict are tracked, and a preceding state of the SAT solver is restored by undoing one or more of the assignments. The SAT solver's search procedure requires the xor-reasoning module to do its computation in an iterative and backtrackable fashion. With this in mind, methods for *storing* and *restoring* the state of xor-reasoning module are needed.

In order to prevent the SAT solver from repeating the same assignments, a clausal explanation is computed by analyzing how the values of variables were inferred and then the explanation is stored as a part of the problem description, so a method to *explain* an inconsistent valuation is included. When the SAT solver analyzes a conflict, it needs to find out how a literal was inferred if it was not added as a decision literal, so literals inferred by the xor-reasoning module need clausal explanations, too.

After the SAT solver has assigned all variables of the cnf-part, the remaining constraints consist only of xor-clauses and Gaussian elimination can be applied. A method to *solve* the remaining xor-part by Gaussian elimination is provided for this purpose.

The design of the combined approach consists of the following parts:

- the interface of the xor-reasoning module
- the proof system defining the inference rules
- a method for computing clausal explanations for conflicts
- a strategy for integrating the xor-reasoning module to a CDCL SAT solver
- design principles for relevant algorithms

In the following chapters, each part of the design will be addressed in detail.

3 XOR-REASONING MODULE

In this chapter, we define the proof system of the xor-reasoning module, how clausal explanations are computed, order of preference for applying inference rules, how variables occurring only in xor-clauses can be addressed, how clausal explanations can contain redundant literals, and how such redundant literals can be removed. The design of the xor-reasoning module is based on the ideas presented in [21]. Based on the requirements for the xor-reasoning module gathered in Section 2.3, we decided to implement the interface presented in Figure 3. The purpose of the xor-reasoning module is to encapsulate the representation of the conjunction of xor-clauses and the operations on them. The interface provides a means for the SAT solver to submit the problem description, to communicate assumptions about the values of variables, to cancel such assumptions, to retrieve logical consequences

of these assumptions inferred by the xor-reasoning module, to identify which of the current assumptions were used to infer a logical consequence, and to consult whether the conjunction of xor-clauses is no longer satisfiable with respect to the current assumptions.

Method	Description
ADD-BACKJUMP-POINT	adds a backjump point which records the state of the xor-reasoning module
ADD-CLAUSE	adds a new xor-clause to the conjunction of xor-clauses
ASSIGN	assigns a truth value to a variable (for communicating assumptions)
BACKJUMP	restores the state recorded by a previously added backjump point
DEDUCE	computes a logical consequence inferred by the current assumptions if possible
EXPLAIN	calculates a clausal explanation for a logical consequence inferred by the xor-reasoning module
SOLVE	decides the satisfiability of the xor-clauses with respect to the current assumptions using Gaussian elimination

Figure 3: Interface of the xor-reasoning module

3.1 Inference Rules

In this section, we define how the xor-reasoning module computes logical consequences of the conjunction of xor-clauses and the current assumptions. This computation is based on a few inference rules which are described using the following syntax:

$$\text{Name}\frac{X_i \qquad X_j}{X_k}$$

where X_i, X_j and X_k are xor-clauses. The rule definition is interpreted as : Given the xor-clauses X_i and X_j, the xor-clause X_k is a logical consequence of X_i and X_j. The xor-clause X_k is always implicitly transformed to normal form. The *proof system* of the xor-reasoning module consists of seven rules : Introduce, Decide$^+$, Decide$^-$, $\oplus$-Unit$^+$, $\oplus$-Unit$^-$, $\oplus$-Eqv$^+$, and $\oplus$-Eqv$^-$.

The Introduce-rule is used to declare a new xor-clause X from a set of xor-clauses ϕ_{xor} to be used as a premise with other rules. Its counterpart in the implementation is the method ADD-CLAUSE defined in Figure 3. The xor-part of the problem instance is defined using this rule.

$$\text{Introduce}\ \frac{}{X}$$

The rules Decide$^+$ and Decide$^-$ define a unary xor-clause A or $A \oplus \top$ where A is an atom and they are in fact special cases of Introduce-rule. Applications of these rules correspond to invocations of the method ASSIGN defined in Figure 3.

$$\text{Decide}^{+}\ \frac{}{A} \qquad \text{Decide}^{-}\ \frac{}{A \oplus \top}$$

The rules Decide^{+} and Decide^{-} are defined to distinguish the assumptions on the truth values of variables made by the SAT solver. The xor-clauses defined using Introduce are part of the problem description and the xor-clauses defined using Decide^{+} and Decide^{-} are assumptions which may be revoked later. The xor-clauses defined using Decide^{+} and Decide^{-} are *xor-assumptions*. When the xor-reasoning module infers $\bot$ as a logical consequence of the xor-part of the problem being solved and the assumptions made by the SAT solver, the SAT solver needs to undo some of its assumptions. For this the xor-reasoning module needs to compute a reason (a subset of xor-clauses inferred using xor-assumptions) for a logical consequence inferred by the proof system. The xor-clauses of the problem description need not to be communicated back to the SAT solver but the xor-clauses defined as assumptions need to be distinguished. That is why the rules Decide^{+} and Decide^{-} are defined. The rules Introduce, Decide^{+} and Decide^{-} are referred as *New-rules*.

The *Inference-rules* of the proof system are listed in Figure 4 where A, A_1 and A_2 are atoms different from $\top$ and X is a xor-clause. The rules are a slightly modified subset of the rules presented in the paper by Baumgartner and Massacci [6]. We present the rules for xor-clauses in normal form instead of xor-clauses with negative literals like in the rules $\oplus$-Unit and $\oplus$-Eqv in [6]. The Gaussian elimination rule presented in [6] is not included here as one of the *Inference-rules* but provided separately by the method SOLVE which performs the full Gaussian elimination. This separation is done in order have a proof system that can be implemented efficiently. The *Inference-rules* are the device for producing logical consequences of the xor-clauses defined using *New-rules*. The SAT solver can use the xor-reasoning module to produce one logical consequence at a time using the method DEDUCE defined in Figure 3.

$$\oplus\text{-Unit}^{+}\ \frac{A \qquad X}{X\,[A/\top]} \qquad \oplus\text{-Unit}^{-}\ \frac{A \oplus \top \qquad X}{X\,[A/\neg\top]}$$

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

Figure 4: Inference rules

We will next define how the rules of the proof system can be applied together to define a xor-derivation of xor-clauses, and then state some important properties of the proof system.

Definition 2. *Let ϕ_{xor} be a set of xor-clauses. A xor-derivation from ϕ_{xor} is a sequence of xor-clauses $D = X_1, \ldots, X_n$ where each X_k, $1 \leq k \leq n$, is either added using one of* New-rules *or is derived from two xor-clauses X_i, $i < k$ and X_j, $j < k$ using one of the* Inference-rules*. A xor-derivation is a xor-refutation if the last xor-clause is $\bot$. A xor-clause X is inferable from a*

set of xor-clauses ϕ_{xor}, denoted by $\phi_{\text{xor}} \vdash X$, iff there is a xor-derivation from ϕ_{xor} with X as the last xor-clause.

Intuitively, a xor-derivation captures a snapshot of a state of the xor-reasoning module. It contains the xor-clauses that define the problem instance, the xor-clauses defined as assumptions, and the xor-clauses that have been inferred from the preceding xor-clauses. Application of *Inference-rules* is referred to as *xor-propagation*. A xor-derivation is *saturated* with respect to xor-propagation if it is not possible to add new xor-clauses in the xor-derivation. The proof system is sound in the sense that all the xor-clauses that are inferred from a set of xor-clauses ϕ_{xor} are logical consequences of the set ϕ_{xor}.

Lemma 1. *(soundness of* Inference-rules*)*
Let X_i, X_j and X_k be xor-clauses. Assume that X_k is derived from X_i and X_j using one of Inference-rules*. It holds that $\{ X_i, X_j \} \models X_k$.*

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

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

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

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

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

□

Theorem 1. *(soundness of xor-derivations)*
Let $D = X_1, \ldots, X_n$ be a xor-derivation. For each $X_i \in D, 1 \leq i \leq n$, it holds that if X_i was derived using one of Inference-rules*, then*

$$\{ X_1, \ldots, X_{i-1} \} \models X_i$$

Proof. We prove the theorem by induction.

Base case $1 \leq i \leq 2$: X_1 and X_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 X_i was not derived using one of *Inference-rules*, the claim holds for i by definition. Otherwise, X_i was derived from two xor-clauses X_j, $1 \leq j < i$ and X_k, $1 \leq k < i, k \neq j$. By Lemma 1, it holds that $\{ X_j, X_k \} \models X_i$. Under the

assumption that $\phi \models X_j$ and $\phi \models X_k$, it holds that $\phi \models X_i$. By definition of logical consequence, the left hand set of xor-clauses $\{ X_j, X_k \}$ can be augmented by additional clauses without compromising logical consequence. It follows that $\{ X_1, \ldots, X_{i-1} \} \models X_i$. □

Example 2. *An example of a xor-refutation is shown in Figure 5. 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 :* $(b \oplus c \oplus \top)\,[b/\top] \rightsquigarrow \top \oplus c \oplus \top \rightsquigarrow c$.

1.	$a \oplus b \oplus c$	Introduce
2.	$a \oplus b \oplus d \oplus e$	Introduce
3.	$c \oplus d \oplus e$	Introduce
4.	a	Decide^{+}
5.	$b \oplus c \oplus \top$	$\oplus$-Unit$^{+}(1, 4)$
6.	$b \oplus d \oplus e \oplus \top$	$\oplus$-Unit$^{+}(2, 4)$
7.	$c \oplus d \oplus e \oplus \top$	$\oplus$-Eqv$^{+}(5, 6)$
8.	b	Decide^{+}
9.	c	$\oplus$-Unit$^{+}(5, 8)$
10.	$d \oplus e \oplus \top$	$\oplus$-Unit$^{+}(3, 9)$
11.	$d \oplus e$	$\oplus$-Unit$^{+}(7, 9)$
12.	$\bot$	$\oplus$-Eqv$^{+}(10, 11)$

Figure 5: Example derivation of xor-clauses

The proof system is sound. However, it is not complete, meaning that there is a set of xor-clauses ϕ_{xor} and a xor-clause X such that $\phi_{\text{xor}} \models X$ but it is not possible to construct a xor-derivation from ϕ_{xor} that includes X. For instance, the empty xor-clause $\bot$ is a logical consequence of the set of xor-clauses $\phi_{\text{xor}} = \{ (a \oplus b \oplus c), (a \oplus b \oplus c \oplus \top) \}$, meaning that ϕ_{xor} is unsatisfiable. However, none of the *Inference-rules* can be applied to infer new xor-clauses from the xor-clauses $(a \oplus b \oplus c)$ and $(a \oplus b \oplus c \oplus \top)$. However, the proof system is *eventually refutationally complete* meaning that if ϕ_{xor} contains an unary xor-clause (A) or $(A \oplus \top)$ for each variable A occurring in ϕ_{xor}, then it is possible to construct a xor-refutation from ϕ_{xor} if and only if the set of xor-clauses ϕ_{xor} is unsatisfiable.

Possibility of Infinite Derivations. It is worth noting that with the inference rules $\oplus$-Eqv^{+} and $\oplus$-Eqv^{-} it is possible to construct infinite xor-derivations. For instance, consider the set of xor-clauses $\phi_{\text{xor}} = \{(a \oplus b \oplus \top), (b \oplus c \oplus \top), (a \oplus c \oplus \top)\}$ and the infinite xor-derivation from ϕ_{xor} in Figure 6. Infinite xor-derivations from a set of xor-clauses ϕ_{xor} can be avoided when the inference rules $\oplus$-Eqv^{+} and $\oplus$-Eqv^{-} are restricted in the following way : (i) the variable with fewer occurrences in ϕ_{xor} must be selected to be substituted with the other, (ii) in case of a tie, the variable to be substituted is selected using

1. $a \oplus b \oplus \top$ Introduce
2. $b \oplus c \oplus \top$ Introduce
3. $a \oplus c \oplus \top$ Introduce
4. $a \oplus b \oplus \top$ $\oplus$-Eqv$^+(2,3)$
5. $b \oplus c \oplus \top$ $\oplus$-Eqv$^+(1,3)$
6. $a \oplus c \oplus \top$ $\oplus$-Eqv$^+(1,2)$
7. ...

Figure 6: Infinite xor-derivation

any fixed ordering, and (iii) once a xor-clause has been used as a premise when applying one of the inference rules $\oplus$-Eqv$^+$ and $\oplus$-Eqv$^-$, the variable to be substituted must be the same for subsequent substitutions.

3.2 Defining Reason Set for Logical Consequence

In this section, we define another representation for xor-derivation - a graph that captures in a better way which parts of the xor-derivation from a set of xor-clauses ϕ_{xor} are needed to infer a logical consequence of ϕ_{xor}. The aim here is to find a way to succinctly define the reason for a logical consequence of ϕ_{xor}. For this, we define how the graph can be partitioned in two parts such that the latter contains logical consequences of the former, and then how this partitioning can be used to define the reason for a logical consequence.

A *directed graph* is a pair $\langle V, E \rangle$ where V is the set of nodes of the graph and the relation $E \subseteq V \times V$ defines which nodes are connected by an edge. A pair $\langle s, t \rangle \in E$ 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 $D = X_1, \ldots, X_n$ be a xor-derivation, the function $rule(X_i)$ gives the rule used to introduce or derive the xor-clause X_i in D, the function $p(X_i)$ gives the set $\{ X_j, X_k \}$ iff the xor-clause X_i is derived from X_j and X_k in D using one of the inference rule, and the function $numDecision(X_i)$ gives the number of xor-assumptions in D up to and including X_i, i.e., the size of the set $\{ X_m \mid X_m \text{ is in } D, m \leq i, r(X_m) \in \{ \text{Decide}^+, \text{Decide}^- \} \}$.

Definition 3. *Given a xor-derivation $D = X_1, \ldots, X_n$, the implication graph of D is a labeled directed acyclic graph $G = \langle V, E, L, r, d \rangle$, where:*

- $V = \{ v_1, \ldots, v_n \}$ *is the set of nodes*
- $L(v_i) = X_i$ *is the labeling function giving the xor-clause X_i of the node v_i*
- $E = \{ \langle v_a, v_b \rangle \mid v_a \in V, v_b \in V, L(v_a) \in p\,(L(v_b)) \}$ *is the set of edges*
- $r : V \to \{$ *Introduce, Decide$^+$, Decide$^-$, $\oplus$-Unit$^+$, $\oplus$-Unit$^-$, $\oplus$-Eqv$^+$, $\oplus$-Eqv$^-$*$\}$ *is defined as follows:* $r(v_i) = rule(L(v_i))$

- $d : V \to \mathbb{N}$ *is defined as follows:*

$$d(v_i) = \begin{cases} 0 & r(v_i) = \textit{Introduce} \\ \textit{numDecision}(L(v_i)) & r(v_i) \in \{\textit{Decide}^+, \textit{Decide}^-\} \\ \textit{max}\,\{\, d(v_a) \mid \langle v_a, v_i \rangle \in E \,\} & r(v_i) \in \text{Inference-rules} \end{cases}$$

A node of an implication graph is a conflict node if it is the xor-clause $\bot$*. A node with the rule label* $\textit{Decide}^+$ *or* $\textit{Decide}^-$ *is a decision node. A node that has two parent nodes is an inferred node.*

The set of nodes V consists of xor-clauses of the xor-derivation D. The graph has an edge from a xor-clause X_a to a xor-clause X_b iff X_b was derived from X_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 the number $(numDecision)$ of the last xor-assumption in the xor-derivation that is also an ancestor of the node.

Example 3. *The implication graph of the xor-derivation in Figure 5 is shown in Figure 7 where the notation for the contents of a node is*

$$[\, i.\, L(v_i) \,|\, r(v_i) \,|\, d(v_i) \,]$$

where $L(v_i)$ *is the* i*th xor-clause in the xor-derivation. The cut lines (cut 1, cut 2, cut 3) can be ignored for now.*

Implication graph presented here is slightly different from implication graphs typically used to reason about satisfiability of Boolean formulas like in [2] by Aspvall et al. where each literal occurring in the Boolean formula has a unique corresponding node in the implication graph. There is an edge from a literal to another literal if the truth value of the former is used to imply the truth value of the latter. This kind of implication graph has been used in the conflict analysis procedure of the SAT solver GRASP [32]. The essential difference to the implication graph presented here is that the nodes contain complete xor-clauses and not just literals. Also, there may be nodes with the same xor-clause.

Each xor-clause of an implication graph that is inferred using one of *Inference-rules* has two incoming edges from the xor-clauses it was inferred from. Exploiting the adjacency relation of an implication graph, the nodes of an implication graph can be partitioned into two sets in such a way that the xor-clauses of the second set are logical consequences of the xor-clauses of the first set. This is defined formally as follows.

Definition 4. *Given a non-empty implication graph* $G = \langle V, E, L, r, d \rangle$*, a cut is a pair* $\langle V_a, V_b \rangle$ *where* V_a *and* V_b *are sets of nodes such that* $V = V_a \cup V_b$ *and* $V_a \cap V_b = \emptyset$*. The premise part of the cut* V_a *contains at least the nodes with no incoming edges and the consequent part* V_b *contains at least the nodes with incoming edges and no outgoing edges.*

When defining a reason for a logical consequence of a set of xor-clauses, we take into consideration only the xor-clauses having outgoing edges that

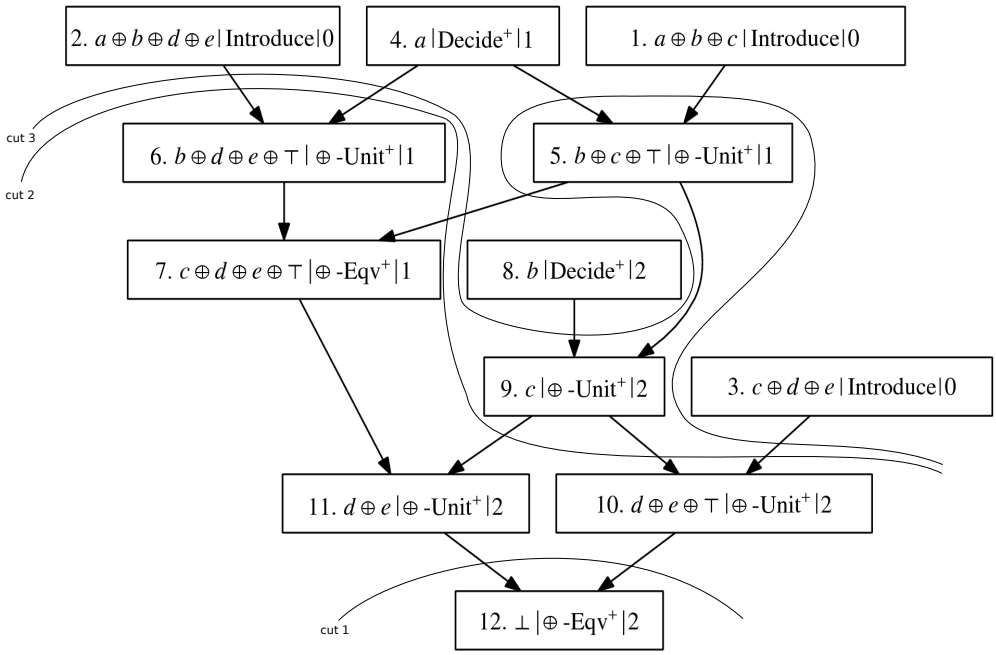


Figure 7: Example implication graph

"cross the cut boundary", that is, have direct successors in the part of the implication graph defined by the consequent 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 consequent part of a cut.

The set of xor-clauses that have direct successors in the consequent part of the cut is split into two sets : the reason set and the supporting set. The supporting set is a subset of the xor-part of the original problem instance and the reason set contains xor-clauses that (i) added using Decide$^+$ or Decide$^-$, or (ii) are derived from other xor-clauses using one of *Inference-rules*. This is defined formally as follows.

Definition 5. *Let $G = \langle V, E, L, r, d \rangle$ be an implication graph and $\langle V_a, V_b \rangle$ be a cut of G. The reason set of the cut $\langle V_a, V_b \rangle$ is a set of xor-clauses ϕ_{rsn} and the supporting set of the cut $\langle V_a, V_b \rangle$ is a set of xor-clauses ϕ_{supp} where:*

$$\begin{aligned} \phi_{\text{rsn}} &= \{\, s \,|\, s \in V_a, r(s) \neq \textit{Introduce}, \exists t \in V_b : \langle s, t \rangle \in E \,\} \\ \phi_{\text{supp}} &= \{\, s \,|\, s \in V_a, r(s) = \textit{Introduce}, \exists t \in V_b : \langle s, t \rangle \in E \,\} \end{aligned}$$

A reason set is cnf-compatible if all xor-clauses in the reason set are of the type (a) or $(a \oplus \top)$, where a is a variable. A cut is cnf-compatible if its reason set is cnf-compatible, and a node is cnf-compatible if its xor-clause has at most one variable. A reason set is a conflict set if $\bot \in V_b$.

If the xor-reasoning module infers the empty xor-clause $\bot$ as a logical consequence of a set of xor-clauses ϕ_{xor}, then the xor-reasoning module is in a conflicting state, meaning that the partial truth assignment $\tilde{M}$ containing the assumptions made by the SAT solver cannot be extended to a model for

ϕ_{xor}. In this case, some of the assumptions have to be revoked and different assumptions picked for further search. This is done by restoring a previously recorded state of the xor-reasoning module identified by a *backjump point* using the method BACKJUMP. In order to prevent the SAT solver from making the same assumptions again and entering the same conflicting state, the assumptions that were used to infer the empty xor-clause $\bot$ are identified and then the particular combination of them is marked as forbidden. This is done by constructing an implication graph of the derived xor-clauses and then computing a cut whose consequent part contains the empty xor-clause $\bot$. The reason set of the cut contains the assumptions needed to infer the empty xor-clause $\bot$. The reason set, which is a conflict set, too, is then converted to an or-clause that the SAT solver stores in its database of learned clauses. For this translation to be possible, the conflict set has to be cnf-compatible so that its negation has a compact logically equivalent representation as an or-clause. The SAT solver retrieves this or-clause using the method EXPLAIN.

Definition 6. *Let ϕ be the conjunction of xor-clauses in a cnf-compatible conflict set ϕ_{conf}. The conflict clause of the conflict set ϕ_{conf} is an or-clause C_{conf} which is logically equivalent to $\neg\phi$.*

Recall that the SAT solver is searching for a model for a cnf-xor-formula $\phi = \phi_{\text{or}} \wedge \phi_{\text{xor}}$. The purpose of conflict clause is to prune the search space of the SAT solver by transferring a portion of the xor-part ϕ_{xor} to the cnf-part ϕ_{or}. It is important that a conflict clause C_{conf} never removes any models for ϕ_{xor}, so the formula $\phi_{\text{xor}} \wedge C_{conf}$ must be logically equivalent to ϕ_{xor}. Formally this is expressed as follows.

Theorem 2. *Let $D = X_1, \ldots, \bot$ be a xor-refutation, $G = \langle V, E, L, r, d \rangle$ the implication graph of D, $\langle V_a, V_b \rangle$ a cut of the implication graph G , ϕ_{conf} the conflict set of the cut $\langle V_a, V_b \rangle$, ϕ_{supp} the supporting set of the cut $\langle V_a, V_b \rangle$, and C_{conf} the conflict clause of the conflict set ϕ_{conf}. It holds that $\phi_{\text{supp}} \models C_{conf}$.*

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

A non-cnf-compatible conflict set ϕ_{conf} can be used to store the reason for a conflict, too, but it cannot be converted to a conflict clause. A cnf-formula ϕ_{or} that is logically equivalent to $\neg\phi_{\text{conf}}$ can always be constructed but in the worst case, the cnf-formula ϕ_{or} can be exponentially longer than ϕ_{conf}. Instead of storing the whole reason for a conflict in the SAT solver's database of learned clauses, another option would be to store the negation of reason set partially in the xor-reasoning module, but studying this in detail is left for future work.

As we are to use only cnf-compatible conflict sets, the correctness of the decision procedure relies on their existence. The following theorem states that a cnf-compatible conflict set can always be formed from the given xor-assumptions.

Theorem 3. *Let $D = X_1, \ldots, \bot$ be a xor-refutation, $\phi_{\text{premises}} \subseteq D$ be the set of xor-clauses introduced using Introduce-rule, $\phi_{\text{decide}} \subseteq D$ be the set of xor-clauses introduced using Decide$^+$- or Decide$^-$-rules, $\phi_{\text{derived}} \subseteq D$ be the set of xor-clauses derived using* Inference-rules*, and $G = \langle V, E, L, r, d \rangle$ be the implication graph of D. There is a cut $\langle V_a, V_b \rangle$ of G such that the conflict set ϕ_{conflict} of the cut $\langle V_a, V_b \rangle$ is a subset of the set ϕ_{decide}.*

Proof. For all nodes $t \in V$, it holds that if $L(t) \in \phi_{\text{premises}}$ or $L(t) \in \phi_{\text{decide}}$, then there are no incoming edges to t. All nodes t such that $L(t) \in \phi_{\text{derived}}$ have incoming edges, so they can be in the second part V_b of the cut. Therefore, it is possible to define a cut $\langle V_a, V_b \rangle$ such that $(\phi_{\text{premises}} \cup \phi_{\text{decide}}) = \{ L(v) \mid v \in V_a \}$ and $\phi_{\text{derived}} = \{ L(v) \mid v \in V_b \}$. By definition, a conflict set never contains any xor-clauses in ϕ_{premises} and all xor-clauses of a conflict set are selected from the first part V_a of a cut. It follows that $\phi_{\text{conflict}} \subseteq \phi_{\text{decide}}$. □

A xor-derivation constructed in such a way that it is always saturated before using the rules Decide$^+$ or Decide$^-$ is said to be *fully saturated*. With a fully saturated xor-derivation it can be argued that the last xor-assumption is the triggering premise for the derivation of all the xor-clauses that follow in the xor-derivation until the next xor-assumption. The xor-clauses that come after a xor-assumption X_a until the next xor-assumption are *xor-consequences* of X_a. Considering reasons for a xor-consequence X_c of a xor-assumption X_a, in some cases, some of the xor-consequences of X_a are enough by itself to infer the xor-consequence X_c if added to the xor-derivation instead of the xor-assumption X_a. These singular xor-clauses are potentially useful when defining succinct reason sets for a logical consequence inferred by the xor-reasoning module. Formally they are defined as follows.

Definition 7. *Let $D = X_1, \ldots, X_n$ be a fully saturated xor-derivation, X_{decide} in D be a xor-clause introduced using Decide$^+$ or Decide$^-$, X in D be a xor-clause defined after X_{decide}, $G = \langle V, E, L, r, d \rangle$ be the implication graph of D, $v_{decide} \in V, v \in V$ be the nodes corresponding to X_{decide} and X, respectively. If there is a path from v_{decide} to v that does not include any nodes whose rule label is Decide$^+$ or Decide$^-$, each node that belongs to all paths from v_{decide} to v is a unique implication point (UIP) for v. The unique implication point that has the shortest distance to v is the first unique implication point.*

Given a unique implication point v_{uip} for some node v in an implication graph, a number of cuts such that v_{uip} is in the premise part of the cut and v in the consequent part of the cut can be defined. These cuts can then be used to define reason sets. The following definition provides a shorthand for discussing reasons sets defined in this way.

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

Example 4. *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 $\{ (x_4 \oplus x_5 \oplus \top), (x_4 \oplus x_5) \}$. *The cut 2 defines the conflict set* $\{ x_1, (x_2 \oplus x_3 \oplus \top), x_3 \}$ *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 cnf-compatible conflict set* $\{ x_1, x_2 \}$ *is* $\neg x_1 \vee \neg x_2$.

3.3 Computing Reason Set for Logical Consequence

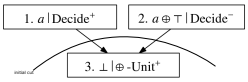
In this section, we define two methods for efficiently computing a reason set for a logical consequence inferred by the xor-reasoning module. The first method computes a cnf-compatible reason set and the second method a cnf-compatible reason set that also contains the first unique implication point for the logical consequence.

Recall that a xor-derivation records a state of the xor-reasoning module. The method we present for computing a reason for a logical consequence is based on traversing the implication graph of the xor-derivation until a suitable cut is found. By suitable cuts we mean cnf-compatible cuts in general, but additional requirements will be introduced for some cases. In most cases, there are several suitable cuts. The quality of a cut can be characterized by how much the corresponding or-clause speeds up the search of the SAT solver when the or-clause is stored and/or used when deriving a conflict clause. Due to the heuristic nature of the search method of the SAT solver, it is highly non-trivial to estimate the quality of a cut. Another issue to consider is the time spent in computing a cut. Also, branching heuristics of the SAT solver and other implementation details may affect the effectiveness of a strategy for defining a cut. Therefore, only the performance of a complete search method can be evaluated.

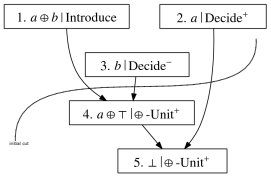
According to the empirical study conducted to evaluate the effectiveness of different learning schemes (strategies for defining cuts of an implication graph) of conflict-driven SAT solvers by Zhang et al., the *1-UIP learning scheme* (first UIP) performed best in the comparison [38]. The implication graph presented in the paper is slightly different from the one defined for xor-derivations in this report. The implication graph of a xor-derivation contains xor-clauses that are not necessarily unique while the implication graph in [38] contains at most one node for each variable. Another importance difference is that the conflict clause defined by a cut has to be *asserting* in the SAT solver, meaning that the conflict clause has to force the truth value of a variable to flip. The conflict clause computed by the xor-reasoning module does not have this requirement because the SAT solver will perform its own conflict analysis starting from the conflict clause returned by the xor-reasoning module in order to get an asserting conflict clause. However, the similarities of the implication graphs still justify studying the effectiveness of the 1-UIP learning scheme adapted in the context of the xor-reasoning module.

Let us consider first the general case of non-fully saturated xor-derivations. Unique implication points are not defined for implication graphs of non-fully saturated xor-derivations, so we are to settle for cnf-compatible cuts. Determining a strategy for finding a generally well-performing cnf-compatible cut is a subject for a detailed study on its own, so we guided our design decisions

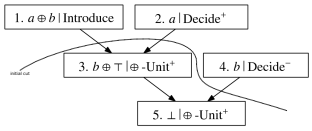
based on an intuition: i) when computing a reason for a logical consequence, the fewer inference steps are needed to derive the logical consequence from a reason set, the better the reason set is, and ii) the less time is spent in computing a reason set, the more efficient the resulting search method is. Therefore, we decided to traverse the implication graph as little as possible when computing a reason set and accept the first cnf-compatible cut. Considering the number of inference steps needed to derive the logical consequence from possible cnf-compatible reason sets, the xor-derivation of the logical consequence from the reason set of the first cnf-compatible cut is the shortest.



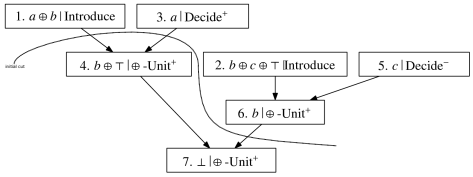
1. Both decision nodes



2. Decision node before inferred node



3. Inferred node before decision node



4. Both inferred nodes

Figure 8: Implication graphs demonstrating initial cuts

The cut is computed by starting with an *initial cut* whose consequent part contains only the node of the implication graph corresponding to the logical consequence that a reason set is being computed for. Figure 8 shows four possible initial cuts whose consequent part contains a conflict node:

1. In the first case, the both parent nodes of the conflict node are decision nodes and the cut cannot be expanded further. The corresponding conflict set contains two xor-clauses (a) and $(a \oplus \top)$ where a is a variable. The conflict set does not have any models so the conflict clause is a tautology and cannot prune the search space of the SAT solver. This case is prevented by forbidding assigning a variable with two different truth values.

2. In the second case, the first parent node of the conflict node is a decision node added before the second parent node which is an inferred node. In this case, the cut can only be expanded through the inferred node. In practice, if xor-propagation is applied in a chronological order with respect to added decision nodes, this case does not occur. For instance, a xor-clause $b \oplus \top$ would have been inferred from the nodes 1 and 2 before inferring $a \oplus \top$ from the nodes 1 and 3.

3. In the third case, the first parent node of the conflict node is an inferred node added before the second parent node which is a decision node. The cut is again expanded through the inferred node leaving the decision node in the conflict set. In practice, this case does not occur because the xor-reasoning module communicates the xor-implied literal (the xor-clause in node 3) to the SAT solver before the second decision node is added so the SAT solver will not add a conflicting xor-assumption. Only if the xor-reasoning module is used without taking xor-implied literals into account, this case may occur.

4. In the fourth case, both parent nodes of the conflict node are inferred nodes. In principle, the cut could be expanded through either parents. In fully saturated xor-derivations, the parent node added later is a unique implication point so it is left in the premise part of the initial cut. When the SAT solver takes xor-implied literals into account, this is the only way to a conflict node can be inferred in the xor-reasoning module.

The cut is modified by moving recursively non-cnf-compatible nodes of the premise part at the boundary of the cut to the consequent part until a cnf-compatible cut is obtained. This can be effectively done by performing a depth-first search on the implication graph and stopping the recursion at cnf-compatible nodes. Visiting the same nodes multiple times is avoided by storing a "timestamp" counter in each node. Each time before a depth-first search is initiated, a global counter indicating the timestamp of the search is incremented. The search is branched to a node only if its timestamp value is smaller than the timestamp of the current search, and then it is set to the value of the global timestamp counter. Algorithm 1 gives a detailed explanation of the process of computing a cnf-compatible reason set for a logical consequence. Its running time is proportional to the number of nodes in the consequent part of the resulting cut.

Algorithm 1 DFS-COMPUTE-REASON-SET($node$)

```
1: SET-TIMESTAMP(node, current_time)
2: if IS-CNF-COMPATIBLE(node) then
3:    ADD-TO-REASON-SET(node)
4: else
5:    for all parent ∈ PARENT-NODES-OF(node) do
6:       if TIMESTAMP-OF(parent) < current_time then
7:          DFS-COMPUTE-REASON-SET(parent)
```

If a xor-derivation is fully saturated, unique implication points are defined, and it is possible to define a cnf-compatible reason set that contains the first (cnf-compatible) unique implication point for a logical consequence inferred by the xor-reasoning module. Finding the first unique implication point for the logical consequence makes computing a reason set slightly more complicated. The depth-first search method presented above cannot be directly applied to this case because it cannot be determined whether a node is the first unique implication point by only inspecting the node itself. The first unique implication point is a node whose decision level is equal to the decision level

of the node the reason set is being computed for (the node corresponding to the logical consequence). The resulting reason set should not contain other nodes with the same decision level except the first unique implication point. This information can be exploited when computing the reason set. Like the method based on depth-first search, the method we present here starts with an initial cut that contains only the logical consequence in the consequent part, and then expands the cut by moving nodes from the premise part to the consequent part until the corresponding reason set is cnf-compatible and contains the first unique implication point. The difference with the method that computes the first cnf-compatible cut is the order in which nodes are moved from the premise part to the consequent part of the cut. If nodes are moved from the premise part to the consequent part in the opposite order they are in the xor-derivation, the first unique implication point can be easily recognized. Provided that the cut is expanded in such a way that the nodes are checked in the opposite order their xor-clauses are in the xor-derivation, the first unique implication point is found when the reason set of the cut has exactly one node whose decision level is equal to the decision level of the node corresponding to the logical consequence. The corresponding cut is called *first-uip-cut*. The reason set has to be cnf-compatible, too, so actually the algorithm gives the first cnf-compatible unique implication point, which is not necessarily the first UIP. Algorithm 2 gives a detailed explanation of the method. Its running time is proportional to the number of nodes in the consequent part of the resulting cut, but the use of a priority queue adds a computational overhead factor proportional to the logarithm of the size of the implication graph. If the number of nodes in the implication graph is relatively low, the nodes can be stored in a linear data structure in the order they are added in the implication graph. Then, a linear scan can be used to compute the cut without the use of priority queue like done in minisat [12] by Eén and Sörensson. The method is conceptually equal to the conflict analysis procedure in the SAT solver minisat.

Algorithm 2 COMPUTE-FIRST-UIP-REASON-SET($node$)

```
 1: set ← { node }
 2: to_check ← NEW-PRIORITY-QUEUE()
 3: ADD-TO-QUEUE(to_check, node)
 4: while (not IS-CNF-COMPATIBLE(set)
              or NODES-WITH-SAME-DECISION-LEVEL(set, node) > 1) do
 5:    next ← EXTRACT-MAX(to_check)
 6:    if (not IS-CNF-COMPATIBLE(next)
          or (HAS-SAME-DECISION-LEVEL(next, node)
                and NODES-WITH-SAME-DECISION-LEVEL(set, node) > 1))
       then
 7:       set ← set\ { next }
 8:       for all parent ∈ PARENT-NODES-OF(next) do
 9:          set ← set ∪ { parent }
10:          ADD-TO-QUEUE(to_check, parent)
```

3.4 Prioritizing Inference Rules

The proof system does not specify in which order the inference rules are to be applied if several rules are applicable. In this section, we show that the order of preference for applying inference rules makes a difference when defining cnf-compatible reason sets for logical consequences inferred by the xor-reasoning module. If the inference rules $\oplus$-Unit$^+$ and $\oplus$-Unit$^-$ are not prioritized over the rules $\oplus$-Eqv$^+$ and $\oplus$-Eqv$^-$, the resulting implication graphs may contain more non-cnf-compatible nodes. When computing a cnf-compatible reason set, antecedent nodes are traversed until a cnf-compatible cut is found. If the implication graph contains more non-cnf-compatible nodes, then more nodes may be traversed resulting in a cnf-compatible reason set with more xor-clauses.

Example 5. *Figure 9 contains an implication graph of a non-fully saturated xor-refutation from a set of xor-clauses* $(a \oplus d \oplus e)$, $(a \oplus b \oplus e \oplus \top)$, $(a \oplus b \oplus g \oplus \top)$, $(c \oplus f \oplus g \oplus \top)$, *and* $(c \oplus d \oplus f)$, *and three xor-assumptions* (a), $(b \oplus \top)$, *and* $(c \oplus \top)$. *The first cnf-compatible cut (cut 1) defines the conflict set* $\{ (a), (c \oplus \top), (e), (f) \}$. *Figure 10 contains a similar implication graph of a non-fully saturated xor-refutation from the same set of xor-clauses and xor-assumptions as above. The first cnf-compatible cut (cut 1) defines the conflict set* $\{ (c \oplus \top), (f \oplus \top), (g) \}$.

The first fifteen nodes in the implication graphs are identical. For the sixteenth node, it is possible to derive the xor-clause $(e \oplus f)$ *using the rule* $\oplus$*-Eqv*$^-$ *or the xor-clause* (d) *using the rule* $\oplus$*-Unit*$^+$. *Depending on the choice of the rule, the conflict set either contains three or four xor-clauses. This happens because the inference rules* $\oplus$*-Unit*$^+$ *and* $\oplus$*-Unit*$^-$ *always make xor-clauses shorter and eventually produce unary xor-clauses which are cnf-compatible. Assuming that the cnf-compatible nodes are uniformly distributed in the implication graph, the more cnf-compatible nodes there are in the implication graph, the less nodes the first cnf-compatible cut contains. The inference rules* $\oplus$*-Eqv*$^+$ *and* $\oplus$*-Eqv*$^-$ *may perform variable substitution without shortening xor-clauses. This may cause the first cnf-compatible to be further away from the (conflict) node being explained and to contain more nodes.*

While not formally proven, we suspect that it is beneficial in the sense of acquiring smaller reason sets on average to prefer the inference rules $\oplus$-Unit$^+$ and $\oplus$-Unit$^-$ over the inference rules $\oplus$-Eqv$^+$ and $\oplus$-Eqv$^-$ in cases where both types of inference rules are applicable. In our preliminary empirical tests, we noticed a significant speedup in the solving times when this kind of order of preference was taken into use.

3.5 XOR-Implied Literals

In this section, we define an important class of logical consequences inferable by the xor-reasoning module : unary xor-clauses. They can be represented as literals and communicated back to the SAT solver. The unit propagation routine of the SAT solver can then continue and possibly deduce new literals. This way the constraints in the xor-part can be exploited to guide the

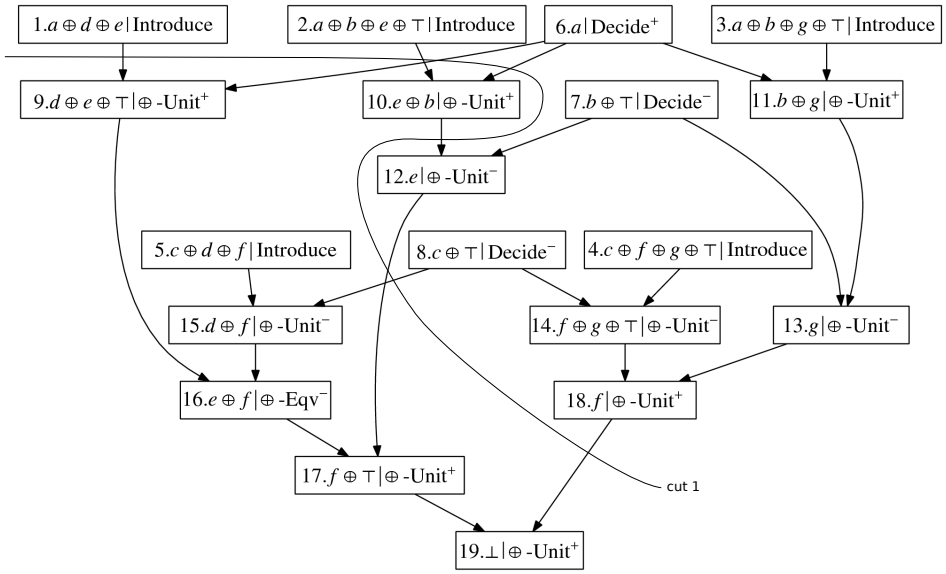


Figure 9: Implication graph without prioritizing unit propagation

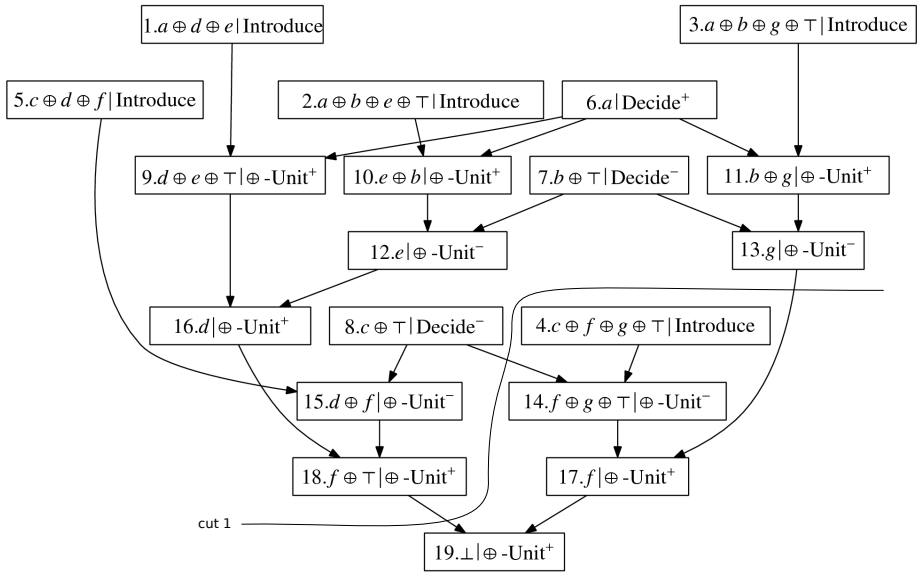


Figure 10: Implication graph when unit propagation is prioritized

1.	$a \oplus b \oplus c$	Introduce
2.	$c \oplus d \oplus \top$	Introduce
3.	a	Decide$^+$
4.	$b \oplus c \oplus \top$	$\oplus$-Unit$^+(1, 3)$
5.	b	Decide$^+$
6.	c	$\oplus$-Unit$^+(4, 5)$
7.	d	$\oplus$-Unit$^+(2, 6)$

Figure 11: Xor-derivation including xor-implied literal

search on the cnf-part. We discuss how such literals can be made logical consequences of the cnf-part, too, so that when the SAT solver derives a conflict, it can find which decision literals were used in the derivation of the conflict.

Definition 9. *Let $D = X_1, \ldots, X_n$ be a xor-derivation. A xor-implied literal is a xor-clause A or $A \oplus \top$ in D not added using* New-rules*, where A is an atom.*

Example 6. *Consider the xor-derivation in Figure 11. Suppose the cnf-xor-formula in question is $\phi = (\neg a \vee c \vee \neg d) \wedge (\neg b \vee \neg c \vee d) \wedge (b \vee \neg c \vee \neg d) \wedge (a \oplus b \oplus c) \wedge (c \oplus d \oplus \top)$. In this example, assume that the SAT solver starts the search by picking the decision literal a which is then propagated to the xor-reasoning module as a xor-assumption. At this point the xor-reasoning module cannot infer any unary xor-clauses. After the second decision literal b is propagated to the xor-reasoning module, two xor-implied literals c and d are propagated back from the xor-reasoning module. The truth assignment $\{ a, b, c, d \}$ is a model for the cnf-xor-formula. Without the xor-implied literals, the SAT solver would have had to pick another decision literal. If this would have been $\neg c$ or $\neg d$, then a xor-conflict would have occurred later when the third decision literal would have been propagated to the XOR module. By propagating facts in both directions (from cnf-part to xor-part and back) fewer decision literals are needed.*

From the point of view of the SAT solver, a xor-implied literal that is propagated back is not really implied because it is not a direct logical consequence of the cnf-part and the assumptions made so far. If it were, then the SAT solver would have already deduced it on its own without the help of the xor-reasoning module. As the xor-implied literal cannot be justified by the cnf-part and the decisions made so far, it would have to be added as an assumption. This certainly is not optimal because by doing this the connection between previous assumptions and the xor-implied literal would be lost. In order to make the xor-implied literal also an implied literal in the SAT solver, we have to supply some kind of justification along with the xor-implied literal. The most straightforward way to do this by an or-clause that enables the SAT solver to deduce the xor-implied literal on its own given the current decision literals.

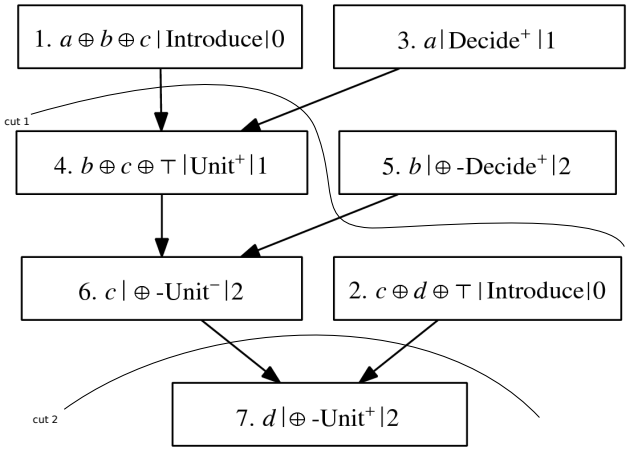


Figure 12: Implication graph with xor-implied literals

Often only a subset of the xor-assumptions is needed to deduce a xor-implied literal. Respectively, a *justifying or-clause* may contain only a subset of all xor-assumptions so far. If the SAT solver stores such a justification as a learned clause in the cnf-part, it can use the stored clause to deduce the same implied literal in similar situations when the search progresses. By similar situations we mean here that all but one of the literals in the justifying or-clause have their negations satisfied by the current partial truth assignment so that the unit propagation can infer the implied literal.

Example 7. *The implication graph of the xor-derivation in Figure 11 is shown in Figure 12. The cut 1 defines the reason set* $\{ a, b \}$ *for the xor-implied literal* c*. The cut 2 defines the reason set* $\{ c \}$ *for the xor-implied literal* d*. We could also choose the cut 1 as a starting point when defining the reason set for the implied literal* d*. So even though two xor-assumptions* a *and* $b \oplus \top$ *were needed to derive the implied literal* d*, a more succinct reason set* $\{ c \}$ *can be given for it. The justifying or-clauses of the xor-implied literal* d *are* $d \vee \neg a \vee \neg b$ *and* $d \vee \neg c$*.*

3.6 XOR-Internal Variables

In this section, we discuss three possible strategies for solving cnf-xor-formulas involving variables that occur only in the xor-part.

A cnf-xor-formula $\phi = \phi_{\text{or}} \wedge \phi_{\text{xor}}$ may contain variables that only occur in the xor-part ϕ_{xor}. We call such variables *xor-internal*. The variables that occur in both parts ϕ_{or} and ϕ_{xor} are referred to as *xor-shared*. For instance, the cnf-xor-formula $\phi = (\neg a \vee b) \wedge (\neg b \vee \neg c) \wedge (a \oplus d) \wedge (b \oplus d) \wedge (d \oplus e) \wedge (b \oplus d \oplus e)$ has xor-internal variables d and e and the variables a and b are xor-shared. When deciding the satisfiability of the cnf-xor-formula ϕ, the SAT solver may find a partial truth assignment $\tilde{M}$ such that $\tilde{M} \models \phi_{\text{or}}$, and then ask the xor-reasoning module if the same holds for the xor-part ϕ_{xor} as well, that is, whether $\tilde{M} \models \phi_{\text{xor}}$, by communicating the literals $l_1, \ldots, l_n \in \tilde{M}$ as xor-assumptions to the xor-reasoning module. Due to the existence of xor-internal variables, it may be that $\tilde{M}$ does not contain literals involving xor-

internal variables. It is also possible that there is no partial truth assignment $\tilde{M}_a \supset \tilde{M}$ such that $\tilde{M}_a \models \phi_{\text{xor}}$, meaning that the formula $\phi_{\text{xor}} \wedge l_1 \wedge \cdots \wedge l_n$ is unsatisfiable. However, as the proof system of the xor-reasoning module is not complete and there are no xor-assumptions concerning xor-internal variables, the xor-reasoning module may fail to detect that the xor-part ϕ_{xor} is unsatisfiable with respect to the current xor-assumptions in $\tilde{M}$. The problem can be addressed in three ways:

1. Encapsulating xor-internal variables: When the SAT solver finds a model for the cnf-part, it performs full Gaussian elimination (the method SOLVE of the xor-reasoning module). The satisfiability of the formula $\phi_{\text{xor}} \wedge l_1 \wedge \cdots \wedge l_n$ can be decided using Gaussian elimination. So by doing this the search method is complete, but there is a drawback in this approach. When a conflict is derived (the xor-reasoning module infers $\bot$ as the logical consequence of $\phi_{\text{xor}} \wedge l_1 \wedge \cdots \wedge l_n$), a reason set capturing the causes of the conflict is extracted. The reason set is then converted to a conflict clause and added as a learned or-clause in the cnf-part. At first, it seems reasonable to avoid reason sets that include xor-internal variables because in theory introducing a new variable in the cnf-part doubles the size of the potential search space. The new variable would be almost unconstrained, having occurrence in only one or-clause. Also, translating constraints on xor-internal variables to CNF might convert the xor-part to CNF. In that case the potential benefits of operating on the xor-part as such would be lost. However, if xor-internal variables cannot be returned in or-clauses returned by the method EXPLAIN, the definition of a suitable cut when computing the reason set has to be slightly modified. The cut has to be expanded possibly further until the corresponding reason set is cnf-compatible and contains only occurrences of xor-shared variables (such reason sets always exists due to xor-assumptions). Computing reason sets in this way may yield larger reason sets due to further expanded cuts and fail to exploit the internal structure of the xor-part. For instance, consider the cnf-xor-formula $\phi = (\neg a \vee b \vee \neg d) \wedge (\neg b \vee d) \wedge (a \vee \neg b \vee \neg d) \wedge (a \oplus b \oplus c) \wedge (c \oplus d \oplus \top)$ which has a xor-internal variable c. Suppose the SAT solver submits xor-assumptions a and b. The xor-reasoning module infers two xor-implied literals c and d. The derivation of the two xor-implied literals is shown in Figure 12. As c is xor-internal variable, it is not propagated to the SAT solver. A justification for c is not needed neither. The implied literal d has occurrences in the cnf-part so the SAT solver will benefit from this fact. If c were not a xor-internal literal, a succinct justification for d would be $d \vee \neg c$ which is defined by the cut 2. However, as c is not available, it has to be replaced with its antecedents that justify it. The cut 1 gives a correct reason set $\{ a, b \}$ which when converted to a justification gives $d \vee \neg a \vee \neg b$.

Other disadvantage is that if the Gaussian elimination deems the formula $\phi_{\text{xor}} \wedge l_1 \wedge \cdots \wedge l_n$ as unsatisfiable, without a method for analyzing how the Gaussian elimination derives the xor-clause $\bot$, the conflict clause can only include xor-assumptions $l_1, \ldots, l_n$. In the worst case, if the cnf-part is a tautology and the xor-part is unsatisfiable and crafted in a certain way, the xor-reasoning module has to perform the computationally intensive Gaussian elimination routine 2^n times where n is the number of xor-shared variables.

This may hinder the effectiveness of the search method for some problem instances. Also, according to the study by Järvisalo and Junttila in [19], if the SAT solver is allowed to branch only on a subset of variables, the resulting search method may produce exponentially longer proofs compared to the unrestricted version. Thus, limiting the choice of the SAT solver to branch only on non-xor-internal variables (make assumptions concerning only variables that occur in the cnf-part) may reduce the effectiveness of the search.

2. Exposing xor-internal variables: Treat the xor-internal variables as xor-shared variables. The SAT solver probably assigns truth values to the variables it knows something about first, but eventually it will start assigning xor-internal variables, as well. In this scheme, as the SAT solver can pick decision literals that also involve xor-internal variables, the potential search space is larger when compared to the scheme where only non-xor-internal variables are considered for decision literals. However, this does not necessarily imply that this scheme would perform worse. If xor-internal variables can occur in clauses returned by EXPLAIN, the computation can stop at the first cnf-compatible cut when computing a reason set. Reason sets computed in this way may be more succinct than reason sets without xor-internal variables. Also, xor-implied literals involving xor-internal variables may prevent the need for the SAT solver to pick decision literals concerning xor-internal variables. If justifying or-clauses are stored in the cnf-part for each xor-implied literal, this eventually translates the xor-part to CNF. This can be avoided by not storing the justifying or-clauses for all xor-implied literals, but only for those that the SAT solver uses to derive a conflict. In fact, not even these justifying or-clauses have to be stored after they are used in the conflict analysis of the SAT solver, but as computing a reason set is a computationally relative expensive operation, we decided to store these important or-clauses. Once they are not used in the search, the SAT solver's "forgetting" heuristics remove inactive or-clauses from the database of learned clauses.

3. Eliminating xor-internal variables: Eliminate xor-internal variables from the xor-part before the search by applying Gaussian elimination. A xor-internal variable x can be eliminated from the xor-part by selecting a xor-clause X containing x and then substituting all occurrences of x in the xor-part with the xor-clause $X \oplus x \oplus \top$. For instance, consider the xor-clauses $X_1 = (a \oplus b \oplus g)$, $X_2 = (c \oplus d \oplus g)$, $X_3 = (e \oplus f \oplus g \oplus h)$, and $X_4 = (c \oplus d \oplus h \oplus i)$ where the variable g is xor-internal and other variables are xor-shared. We choose $X_3 \oplus g \oplus \top$ as the "definition" of g and define new xor-clauses $X_1' = X_1\,[g/(X_3 \oplus g \oplus \top)] = (a \oplus b \oplus (e \oplus f \oplus g \oplus h \oplus g \oplus \top)) \rightsquigarrow (a \oplus b \oplus e \oplus f \oplus h \oplus \top)$, $X_2' = X_2\,[g/(X_3 \oplus g \oplus \top)] = (c \oplus d \oplus e \oplus f \oplus h \oplus \top)$, and $X_4' = X_4$. The resulting xor-clauses X_1' and X_2' contain now four variables. Remaining xor-clauses grow longer when xor-internal variables are eliminated. The inference rules of the presented proof system are defined for xor-clauses that have at most two variable occurrences. As more xor-assumptions are needed to infer logical consequences using the inference rules, making xor-clauses longer may reduce the effectiveness of the proof system. With the original xor-clauses X_1, X_2, X_3 and X_4 it is possible to infer a xor-implied literal i from the xor-assumptions (a) and (b). This cannot be

done if the xor-internal variable g is eliminated. Due to the obvious deficiency in this approach, we decided not to test experimentally the effect of elimination of xor-internal variables.

Encapsulating xor-internal variables may reduce the size of the search space effectively, so it is justifiable to evaluate the approach experimentally. However, when xor-internal variables are encapsulated in the xor-reasoning module, they are not returned in reason sets resulting potentially in more inefficient conflict clauses in the SAT solver. Therefore, the approach in which xor-internal variables are exposed should be evaluated experimentally as well. The inference rules of the xor-reasoning module work most effectively when xor-clauses are short, so the approach in which xor-internal variables are eliminated is excluded from the experimental evaluation because xor-clauses grow longer if xor-internal variables are eliminated. Experimental evaluation of elimination of xor-internal variables is left for future work along with a study on how xor-clauses can be preprocessed before the actual search.

3.7 Redundancy in Reason Set

In this section, we discuss the possibility of a reason set containing xor-clauses that are inferable from the xor-part of the problem and the other xor-clauses in the reason set. These xor-clauses are in a way redundant and can be removed from the reason set without affecting the meaning of the reason set. The motivation for this is an intuition that the fewer literals a conflict clause or a justifying or-clause contains, the more useful it is in reducing the amount of computation required in the search done by the SAT solver. It is always possible to define a cut of an implication graph whose reason set does not have any redundant xor-clauses, so the possibility of redundant xor-clauses in a reason set is tied to how a cut is computed. The methods for computing a reason set we have presented may produce reason sets that contain redundant xor-clauses. We will present a method for computing a reason set that produces reason sets without redundant xor-clauses and a method for removing some redundant xor-clauses from a reason set. Redundant xor-clause is formally defined as follows:

Definition 10. *Let $D = X_1, \ldots, X_n$ be a xor-derivation from ϕ_{xor} and ϕ_{reason} be a reason set for some cut of the implication graph of D. A redundant xor-clause in ϕ_{reason} is a xor-clause $X_r \in \phi_{\text{reason}}$ for which holds that $(\phi_{\text{xor}} \cup (\phi_{\text{reason}} \setminus \{ X_r \})) \vdash X_r$. A reason set is minimal if it does not have redundant xor-clauses.*

A redundant xor-clause can be removed from a reason set because it can be inferred from the reason set and the xor-part of the cnf-xor-formula being solved. Note that our definition of redundant xor-clause covers only inferable xor-clauses. Even when a reason set does not contain any redundant xor-clauses, it may contain xor-clauses that are logical consequences of the other xor-clauses in the reason set and the xor-clauses in the xor-part.

Example 8. *Consider the implication graph in Figure 13. The xor-clause $\bot$ is derived from two xor-assumptions (a) and (b) and from the xor-clauses*

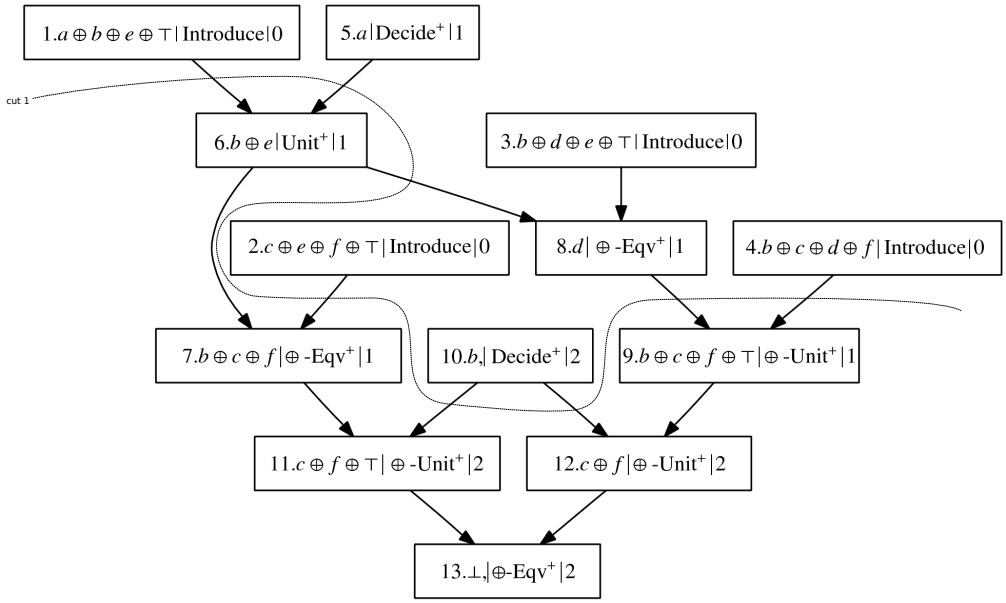


Figure 13: Implication graph with a redundant literal

$(a \oplus b \oplus e \oplus \top)$, $(c \oplus e \oplus f \oplus \top)$, $(b \oplus d \oplus e \oplus \top)$, *and* $(b \oplus c \oplus d \oplus f)$. *The cut 1 defines the reason set* $\{ a, b, d \}$. *As the reason set includes both xor-assumptions* (a) *and* (b), *the xor-clause* (d) *must be redundant. In fact,* (d) *is inferred from the xor-assumption* (a) *and from the xor-clauses* $(a \oplus b \oplus e \oplus \top)$ *and* $(b \oplus d \oplus e \oplus \top$ *so it can be removed from the reason set, which results in a minimal reason set* $\{ a, b \}$.

In order to define a method for computing a reason set free of redundant xor-clauses, we consider reason sets based on fully saturated xor-derivations. A fully saturated xor-derivation from a set of xor-clauses ϕ_{xor} contains all xor-clauses that are inferable from ϕ_{xor} and ordered in a way that xor-consequences in the xor-derivation cannot be inferred without the preceding xor-assumption. So, with fully saturated xor-derivations, assuming that xor-assumptions are not added for xor-clauses already in the xor-derivation, they are not redundant in any reason set for any xor-consequence. Unique implication points are defined for fully saturated xor-derivations. As xor-assumptions (which are unique implication points, too), they have the same important property of not being redundant by definition so we can construct a method based on unique implication points that guarantees reason sets free of redundant xor-clauses.

When computing a reason set for a xor-consequence, the method picks only the unique implication points needed to infer the xor-consequence by traversing the implication graph in the same way as the two methods we have presented but with a different definition for a suitable cut: expanding the cut is stopped when the corresponding reason set contains at most one xor-clause from each decision level. We call such a cut *all-uip-cut*. Algorithm 3 provides a detailed explanation of the method.

If there is a method for computing a reason set without redundant xor-clauses, it is justified to question the use of methods for computing a reason

Algorithm 3 COMPUTE-ALL-UIP-REASON-SET($node$)

```
 1: set ← { node }
 2: to_check ← NEW-PRIORITY-QUEUE()
 3: ADD-TO-QUEUE(to_check, node)
 4: while (not IS-CNF-COMPATIBLE(set)
          or NODES-WITH-SAME-DECISION-LEVEL(set) > 1) do
 5:    next ← EXTRACT-MAX(to_check)
 6:    if (not IS-CNF-COMPATIBLE(next)
         or NODES-WITH-SAME-DECISION-LEVEL(set, next) > 1) then
 7:       set ← set\ { next }
 8:       for all parent ∈ PARENT-NODES-OF(next) do
 9:          set ← set ∪ { parent }
10:          ADD-TO-QUEUE(to_check, parent)
```

set that allow redundant xor-clauses. The following example illustrates that a reason set of an all-uip-cut may contain more xor-clauses than a reason set of a first-uip-cut.

Example 9. *In the implication graph in Figure 14 the two unary xor-clauses* (g) *and* (h) *are on the same decision level. They are both xor-consequences of the xor-assumptions* (a)*,* (b)*, and* (c)*. The first-uip-cut (cut 1) defines the reason set* $\{ h, d, g \}$ *which contains two xor-clauses from the decision level 3. If at most one xor-clause is selected from each decision level, the all-uip-cut (cut 2) defines the reason set is* $\{ a, b, c, d \}$ *which has one xor-clause more than the other reason set.*

However, the opposite is true, as well. The reason set of a first-uip-cut may contain more xor-clauses than the reason set of an all-uip-cut. This is shown in the following example.

Example 10. *The xor-assumption* (a) *in the implication graph in Figure 15 is used to infer two unary xor-clauses* (f) *and* (g)*. The first-uip-cut (cut 1) defines the reason set* $\{ f, g, b \}$ *which contains one xor-clause more than the reason set of the all-uip-cut (cut 2)* $\{ a, b \}$*.*

As it seems to be non-trivial to find the smallest cnf-compatible reason set in the general case, the method that is computationally less expensive would be good candidate to perform well. However, the reason sets of first-uip-cuts may still contain redundant xor-clauses. It is even possible to construct implication graphs of fully saturated xor-derivations whose first-uip-cuts contain an arbitrarily large number of redundant xor-clauses . With this possibility in mind, it is reasonable to explore whether it is feasible to identify which xor-clauses in a reason set are redundant. By performing a dfs-search in the implication graph starting from the parents of a node whose xor-clause is in the reason set, it can be determined if the xor-clause can be inferred from other xor-clauses of the reason set. If a xor-assumption that it is not in the reason set can be reached from the node the search is started from, then the xor-clause of the node is not redundant. Also, as an optimization, the search can be terminated if a node whose index is smaller than the smallest index

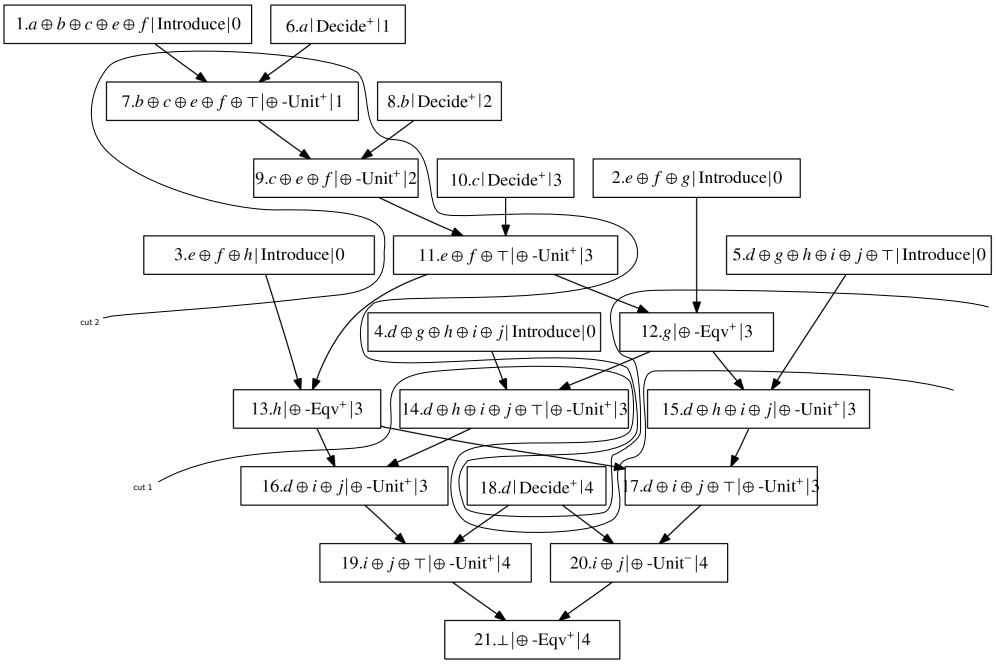


Figure 14: Implication graph where all-UIP-cut (cut 2) is wider than first-UIP-cut (cut 1)

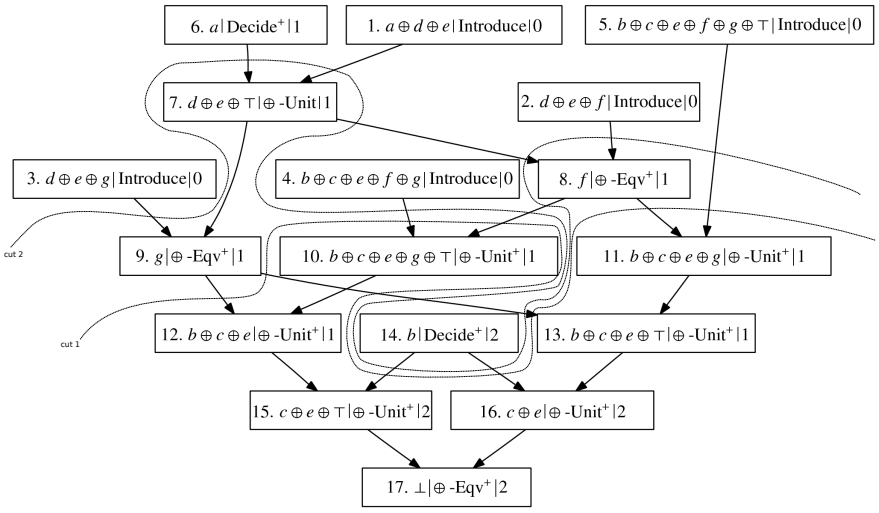


Figure 15: Implication graph where first-UIP-cut (cut 1) is wider than all-UIP-cut (cut 2)

in the reason set. The details of the method for identifying redundant xor-clauses are described in Algorithm 4. Timestamp counters and computed values are stored in the nodes in order to visit only once each node. As it seems to be computationally less expensive to compute first-uip-cuts than all-uip-cuts, and redundant xor-clauses can be eliminated if needed, we decided to leave the implementation of a method that computes reason sets based on all-uip-cuts for future work.

With non-fully saturated xor-derivations it cannot be determined directly by the implication graph if a xor-clause in a reason set is redundant or not. A xor-clause in the reason set could possibly be inferred from fewer xor-assumptions if the xor-derivation were fully saturated. That is why the method for identifying redundant xor-clauses does not work completely with non-fully saturated xor-derivations. However, if it identifies a xor-clause as redundant, then it is redundant and can be removed.

Sörensson and Biere have independently used a similar procedure to minimize learned clauses in their SAT solver minisat and found it effective [34].

Algorithm 4 IS-INFERABLE(set, $node$)

```
 1: SET-TIMESTAMP(node, current_time)
 2: SET-INFERABLE-FLAG(node, false)
 3: if INDEX-OF(node) < SMALLEST-INDEX-IN(set) then
 4:    return false
 5: else
 6:    if IS-IN-REASON-SET(node, set) then
 7:       SET-INFERABLE-FLAG(node, true)
 8:       return true
 9:    else
10:       if IS-XOR-ASSUMPTION(node) then
11:          return false
12:       else
13:          for all parent ∈ PARENT-NODES-OF(node) do
14:             if TIMESTAMP-OF(parent) < current_time then
15:                if IS-INFERABLE(parent) then
16:                   return false
17:             else
18:                if GET-INFERABLE-FLAG(node) = false then
19:                   return false
20:          SET-INFERABLE-FLAG(node, true)
21:          return true
```

4 CNF/XOR INTEGRATION

The xor-reasoning module was designed in a way that it could be easily integrated to an existing SAT solver capable of clause learning. In this chapter, we describe an abstract model of a Conflict Driven Clause Learning Satisfiability Solver (CDCL SAT) and its semantics. We extend the abstract CDCL SAT solver with rules to perform propagation in the xor-part and to

communicate information between the cnf-part and the xor-part. We discuss three possible alternatives for propagation strategies (rule application priorities). The first propagation strategy is based on fully saturated xor-derivations. When unit propagation is saturated in the SAT solver, the literals known by the SAT solver are propagated to the xor-reasoning module one by one. After each propagated literal, all inferable xor-clauses are computed by the xor-reasoning module. In the second propagation strategy, unit propagation in the SAT solver and the literals inferred and assumed by the SAT solver are prioritized over the inference rules of the xor-reasoning module so the xor-reasoning module is used as little as possible. In the third propagation strategy, the SAT solver finds first a model for the cnf-part of the cnf-xor-formula being solved. Only when a model for the cnf-part is found, the literals known by the SAT solver are propagated to the xor-reasoning module. A discussion about strategies for handling xor-implied literals concerning the use of justifying or-clauses in the SAT solver concludes the chapter.

4.1 CDCL SAT Solver

In this section, we define an abstract model of CDCL SAT solver based on the DPLL(T) model by Nieuwenhuis, Oliveras, and Tinelli [28]. The CDCL SAT solver is modelled as a stateful system whose state can be altered with a handful of state transition rules. The state transition rules abstract away implementation details concerning representation of the cnf-formula and modifications on data structures. Until now, we have described the search of a SAT solver as modifications to a partial truth assignment. In order to simplify the definition of the state transition rules, we increase the level of detail of in our model of the CDCL SAT solver. The following definitions give a more accurately description of how a partial truth assignment – the literals known by the solver at a given moment – is represented in our model of the CDCL SAT solver.

Definition 11. ***(trail).*** *A trail $\pi = (l_1^{R_1}, \ldots, l_n^{R_n})$ is a sequence of annotated literals where each $l_i \in \{ l_1, \ldots, l_n \}$ is a literal such that there is an or-clause in ϕ_{or} with an occurrence of l_i, and each reason $R_i \in \{ R_1, \ldots, R_n \}$ is an or-clause in ϕ_{or} or the symbol d which denotes empty reason (the reason for a decision literal).*

Definition 12. ***(satisfiability relation of trail).*** *We define that $\pi \models C$ iff there is a literal l in the or-clause C and also one of the literals of the trail π. To indicate a situation where the trail π cannot be completed to a model for the or-clause C, we define that $\pi \models \neg C$ iff the negation $\neg l$ of each literal l in the or-clause C is in the trail π.*

Definition 13. ***(invariants of trail).*** *For each prefix of π, $\pi_{\text{pfx}} = l_1^{R_1}, \ldots, l_i^{R_i}$, $i < n$, it must hold:*

- *$R_{i+1} = \mathsf{d}$ or otherwise $R_{i+1} = (C \vee l_{i+1})$ and $\pi_{\text{pfx}} \models \neg C$ (valid reasons and reasons in order).*
- *$L = \left\{ l \,\middle|\, l^R \in \pi_{\text{pfx}} \right\}$, $l_{i+1} \notin L$, $\neg l_{i+1} \notin L$ (no same literal nor its negation twice).*

UnitPropagate $$\frac{R = (C \vee l) \in \phi_{\text{or}} \quad \pi \models \neg C}{\pi \leftarrow \pi \,.\, l^R}$$

Decide $$\frac{C \in \phi_{\text{or}} \quad l \in C \quad l \notin \pi}{\pi \leftarrow \pi \,.\, l^{\mathsf{d}}}$$

Fail $$\frac{\neg \exists l : l^{\mathsf{d}} \in \pi \quad C \in \phi_{\text{or}} \quad \pi \models \neg C}{Fail}$$

Backjump $$\frac{\pi = \pi_{\text{s}}.l^{\mathsf{d}}.\pi_{\text{e}} \quad R{=}(C \vee k){\in}\phi_{\text{or}} \quad \pi{\models}\neg R \quad \pi_{\text{s}}{\models}\neg C \quad k \notin \pi_{\text{s}}}{\pi \leftarrow \pi_{\text{s}} \,.\, k^R}$$

Learn $$\frac{\phi_{\text{or}} \models C \quad \forall\, l \in C : \neg l^R \in \pi}{\phi_{\text{or}} \leftarrow \phi_{\text{or}} \cup C}$$

Figure 16: State transition rules of CDCL SAT solver

- *if $R_{i+1} = \mathsf{d}$, then there must not be an or-clause $(C \vee l) \in \phi_{\text{or}}$ such that*

$$(l_1 \wedge \cdots \wedge l_i \models \neg C) \wedge (l \notin \{\, l_1, \ldots, l_i \,\})$$

(unit propagation saturated-invariant)

Definition 14. ***(concatenating trails).*** *A trail can be defined as a concatenation of two parts of trail (one or more elements), for instance $\pi' = \pi \,.\, l^R$ defines the trail π' which has the trail π as a prefix and in addition the annotated literal l^R in the end.*

The trail is like a partial truth assignment but makes the order of the literals significant and also adds information on what grounds the literals are added to the trail (either as decision literals or literals inferred by unit propagation). The state of the CDCL SAT solver is represented by a pair $\langle \phi_{\text{or}}, \pi \rangle$. The rules for valid state transitions are described using following syntax:

$$\text{name of the rule} \; \frac{\text{applicability condition}}{\text{state modification}}$$

For the following definitions of the state transition rules in Figure 16, let C and R be or-clauses, and k, l be literals. For the sake of compactness we use the expression $l^R \in \pi$ to indicate that the literal l^R occurs in the trail π. The expression $l \notin \pi$ means that the literal l (with any associated reason) is not in the trail π.

The state transition rules presented here are basically the ones presented in [28] which also discusses the soundness and completeness of the rules and the finiteness of state sequences. We omitted the rules Forget and Restart because they are not relevant in the context of integrating the xor-reasoning module to the CDCL SAT solver. The rule UnitPropagate is applicable when there is an or-clause that can be used as a premise for unit propagation. It modifies the state of the SAT solver by adding a literal and its reason to the trail. The rule Decide is used to add decision literals in the trail. The

π	Justification	Rule
1. a^{d}	-	Decide
2. b^{R}	(1) $R = \neg a \vee b$	UnitPropagate
	$\phi_{\text{or}} \leftarrow \phi_{\text{or}} \cup \{ (\neg a) \}$	Learn
$\pi \leftarrow \neg a^{R'}$	$R' = C \vee \neg a, C = ()$	Backjump

Figure 17: CDCL SAT solver deduces a conflict

rule Fail is used to conclude a failed search for a model when a conflict can be derived without any decision literals. The rule Backjump is applicable when the trail cannot be completed to a model and the cnf-part contains a *backjump clause* that directs the search to an unexplored part of the search space. The trail is cut to a point where the backjump clause can be used to infer a literal. The rule Learn is used to add backjump clauses that are also logical consequences of the cnf-part. Note that with Decide-rule, it is possible to get a sequence of annotated literals that is not a trail (as it violates unit-propagation invariant).

Example 11. *Assume the CDCL SAT solver is searching for a model for a cnf-formula $\phi_{\text{or}} = (\neg a \vee b) \wedge (\neg a \vee \neg b)$. A part of the search is shown in the trail table in Figure 17. A trail table captures a snapshot of the state of the CDCL SAT solver. Numbering in the trail column (the column with the label "π" records the propagation order of literals). The column labeled "Justification" explains why it is possible to apply the state transition rule mentioned in the column "Rule". In the example the SAT solver picks a decision literal, infers another literal using unit propagation, derives a conflict, learns a conflict clause, and backjumps (cancels the decision).*

4.2 CDCL/XOR SAT Solver

The CDCL SAT solver model can be used to model the functionality of SAT solvers at an abstract level. In this section, we extend this model to introduce CDCL/XOR SAT Solver, a CDCL SAT solver with the xor-reasoning module integrated into it , in order to discuss how the interaction of the SAT solver and the xor-reasoning module can be implemented to solve efficiently cnf-xor-formulas of type $\langle \phi_{\text{or}}, \phi_{\text{xor}} \rangle$. The SAT solver operates on the cnf-part ϕ_{or} and the xor-reasoning module on the xor-part ϕ_{xor}. The search is driven by the SAT solver and the xor-reasoning module is used as a subroutine. The SAT solver propagates the literals in its trail to the xor-reasoning module according to one of the propagation strategies which will be discussed later. A state of the CDCL/XOR SAT solver is a tuple $\langle \phi_{\text{or}}, \phi_{\text{xor}}, \pi_{\text{dpll}}, \pi_{\text{xor}} \rangle$ where ϕ_{or} is the cnf-part, ϕ_{xor} is the xor-part, π_{dpll} is the trail of the SAT solver, and the additional trail π_{xor} is a prefix of π_{dpll} which tracks the literals that have been propagated to the xor-reasoning module.

We introduce some additional state transition rules that model the interaction between the SAT solver and the xor-reasoning module. As before, we define some identifiers used in the definitions below. Let C, R be or-clauses, X be a xor-clause, and k, l be literals. The state transition rules specific to CDCL/XOR SAT solver are presented in Figure 18. The trail

XorPropagate
$$\frac{\pi_{\text{dpll}} = \pi_{\text{xor}} \,.\, l^R \,.\, \pi_{\text{e}}}{\pi_{\text{xor}} \leftarrow \pi_{\text{xor}} \,.\, l^R}$$

XorImply
$$\frac{(\phi_{\text{xor}} \wedge \pi_{\text{xor}}) \models l \quad \phi_{\text{xor}} \models R \quad (\phi_{\text{or}} \cup R) \wedge \pi_{\text{or}} \models l \quad \neg l \notin \pi_{\text{dpll}}}{\phi_{\text{or}} \leftarrow \phi_{\text{or}} \cup R \quad \pi_{\text{dpll}} \leftarrow \pi_{\text{dpll}} \,.\, l^R}$$

XorConflict
$$\frac{\pi_{\text{xor}} \models \neg X \quad X \in \phi_{\text{xor}} \quad \phi_{\text{xor}} \models C \quad \pi_{\text{xor}} \models \neg C}{\phi_{\text{or}} \leftarrow \phi_{\text{or}} \cup C}$$

XorFail
$$\frac{\neg \exists l : l^{\mathsf{d}} \in \pi_{\text{xor}} \quad X \in \phi_{\text{xor}} \quad \pi_{\text{xor}} \models \neg X}{Fail}$$

Figure 18: State transition rules specific to CDCL/XOR SAT solver

π_{dpll}	π_{xor}	Justification	Rule
1. a^{d}			Decide
2. b^R		$R = \neg a \vee b$	UnitPropagate
	3. a^{d}	(1)	XorPropagate
4. $c^{R'}$		$(a) \wedge (a \oplus c \oplus \top) \models c, \quad R' = \neg a \vee c$	XorImply
	5. b^R	(2)	XorPropagate
		$X = a \oplus b, \quad C' = \neg a \vee \neg b$	XorConflict

Figure 19: CDCL/XOR SAT solver deduces a conflict

$\pi_{\text{xor}} = l_1^{R_1}, \ldots, l_n^{R_n}$ is interpreted as a conjunction $(l_1 \wedge \cdots \wedge l_n)$ in the expression $(\phi_{\text{xor}} \wedge \pi_{\text{xor}}) \models l$.

The rule XorPropagate is used to indicate when a literal in the trail of the SAT solver is propagated to the xor-reasoning module (the method ASSIGN of the xor-reasoning module). The rule XorImply adds a xor-implied literal in the trail of the SAT solver along with a justifying or-clause for the xor-implied literal in the cnf-part (the methods DEDUCE and EXPLAIN of the xor-reasoning module). The rule XorImply corresponds to the rule TheoryPropagate presented in [28]. The rule XorConflict is used to add a conflict clause computed by the xor-reasoning module in the cnf-part (the method EXPLAIN of the xor-reasoning module). The rule XorConflict is related to the rules T-Learn and T-Backjump in [28]. The rule XorFail is used to conclude the failed search for a model when a xor-refutation can be constructed from the xor-part without any decision literals. The rule Backjump is otherwise similar to its counterpart in the model of CDCL SAT solver but the trail of the xor-reasoning module is synchronized with the trail of the SAT solver.

The state transition rules that model the interaction of the SAT solver and the xor-reasoning module are used to indicate when and where a particular computation is done in the CDCL/XOR SAT solver. Therefore we can assume that they do not compromise soundness, completeness, termination of the abstract CDCL SAT solver.

Example 12. *Suppose the CDCL/XOR SAT solver is solving the cnf-xor-formula* $(\neg a \vee b) \wedge (\neg a \vee \neg b \vee \neg c) \wedge (a \oplus c \oplus \top) \wedge (a \oplus b)$. *A trail table*

of the search in Figure 19 shows how the xor-reasoning derives a conflict after receiving two xor-assumptions (a) and (b) from the SAT solver. The new column labeled "π_{xor}" indicates the literals that are propagated to the xor-reasoning module.

4.3 Fully Saturated XOR-Propagation

In this section, we will present a propagation strategy that is based on fully saturated xor-derivations. The idea is the same as in *exhaustive theory propagation* presented in [28]. The effectiveness of a propagation strategy is determined by three factors: i) the computational cost of the operations of the xor-reasoning module, ii) the effect the operations of the xor-reasoning module on the length of the search, and iii) the cnf-xor-formula being solved. When other characteristics of the SAT solver are kept unchanged, the optimal propagation strategy (for a certain set of cnf-xor-formulas) is simply the one that makes the SAT solver terminate the search in shortest time on average. Due to the heuristic nature and general applicability of SAT solvers, only the computational cost of the operations of the xor-reasoning module can be accurately measured so we decided to implement multiple propagation strategies and perform empirical experiments in order to determine which propagation strategy performs best.

The *fully saturated xor-propagation* (strategy) works as follows. Unit propagation in the cnf-part is likely to be faster than propagation in the xor-part due to the highly optimized data structures and algorithms of modern SAT solvers, so unit propagation in the cnf-part is preferred whenever possible. When it is no longer possible to do unit propagation in the cnf-part, the literals in the trail are propagated to the xor-reasoning module. All xor-consequences of the xor-assumption (the literal propagated by the SAT solver) are computed and propagated back to the SAT solver before propagating another literal to the xor-reasoning module. Conflicts may occur in three ways: i) unit propagation in the cnf-part results in an unsatisfied or-clause, ii) xor-implied literal makes an or-clause unsatisfied in the cnf-part, and iii) xor-assumptions and propagation in the xor-part results in a conflicting (empty) xor-clause. Fully saturated xor-propagation is interesting due to the possibility to compute reason sets of first-uip-cuts. While it would be possible to use first cnf-compatible-cuts when computing a cut, we decided to leave studying this option for future work in order to emphasize the use of xor-reasoning module and xor-based reasoning in this xor-propagation strategy. The fully saturated xor-propagation strategy can be characterized by associating the following order of preference to the state transition rules (the first applicable rule is applied always): UnitPropagate, XorImply, XorPropagate, Decide.

An overview of the fully saturated xor-propagation search method is shown in Algorithm 5. The search is driven by unit propagation in the cnf-part (line 1, UNIT-PROPAGATE). If a conflict can be derived by only doing unit propagation in the cnf-part, the literals in the trail are not propagated to the xor-reasoning module (lines 4-16 are skipped). Otherwise, the literals inferred by unit propagation in the cnf-part are propagated to the xor-reasoning module one by one (lines 5-6). All xor-consequences of each propagated literal are added to the trail along with their reasons (lines 7-14). If a conflict occurs

without decision literals in the cnf-part or in the xor-part, the cnf-xor-formula has no models (lines 17-18). Otherwise, the SAT solver performs conflict analysis (the method ANALYZE-CONFLICT), adds a conflict clause in the cnf-part, revokes decision literals until the conflict is resolved, and restores the state of the xor-reasoning module to a previously recorded state (lines 19-20). If no conflict is derived and propagation is saturated both in the cnf-part and in the xor-part, a new decision literal is picked if possible (lines 22-23). If there are no unassigned variables and no conflict can be derived, Gaussian elimination is performed to assign remaining unassigned xor-internal variables (the method XOR-SOLVE). If the xor-internal variables can be assigned without conflicts, a model has been found (line 25).

Algorithm 5 FULLY-SATURATED-XOR-PROPAGATION-SEARCH(ϕ_{or}, ϕ_{xor})

```
 1: π_dpll, π_xor ← (), ()
 2: loop
 3:    (π_dpll, confl) ← UNIT-PROPAGATE(φ_or, π_dpll)
 4:    if not confl then
 5:       for each literal l in π_dpll and not in π_xor do
 6:          π_xor ← XOR-ASSIGN(π_xor, l)
 7:          (k_1, ..., k_n) ← XOR-DEDUCE(π_xor, φ_xor)
 8:          if n > 0 then
 9:             for each xor-consequence k ∈ (k_1, ..., k_n) do
10:                C ← XOR-EXPLAIN(k)
11:                if π_dpll ⊨ ¬C then
12:                   confl ← C
13:                   break
14:                else
15:                   π_dpll ← π_dpll . k^C if k ∉ π_dpll
16:             goto 2 /* continue with unit propagation */
17:    if confl then
18:       if not HAS-DECISION-LITERALS(π) then
19:          return unsat
20:       (π_dpll, φ_or) ← ANALYZE-CONFLICT(φ_or, confl, π_dpll)
21:       π_xor ← XOR-BACKJUMP(π_dpll)
22:    else
23:       if HAS-UNASSIGNED-VARIABLES(φ_or, π_dpll) then
24:          π_dpll ← ADD-DECISION-LITERAL(φ_or, π_dpll)
25:       else
26:          confl ← XOR-SOLVE(φ_xor, π_xor)
27:          if confl then
28:             goto 17
29:          return sat
```

Example 13. *While the fully saturated xor-propagation strategy has the attractive property of producing reason sets based on first-uip-cuts, letting the xor-reasoning module to calculate all xor-consequences of each propagated xor-assumption before propagating the next xor-assumption may cause the xor-reasoning module to infer xor-implied literals that are already known by the SAT solver. The trail table in Figure 20 illustrates a state of the CDCL*

π_{dpll}	π_{xor}	Justification	Rule
1. a^{d}		-	Decide
2. b^{R_1}		$R_1 = \neg a \vee b$	UnitPropagate
3. c^{R_2}		$R_2 = \neg b \vee c$	UnitPropagate
	4. a^{d}	(1)	XorPropagate
5. (b^{R_3})		$(a) \wedge (a \oplus b \oplus \top) \models b, \quad R_3 = \neg a \vee b$	XorImply
6. (c^{R_4})		$(a) \wedge (a \oplus c \oplus \top) \models c, \quad R_4 = \neg a \vee c$	XorImply

Figure 20: Xor-reasoning module infers already known xor-implied literals

SAT solver operating on the cnf-xor-formula $\phi = (\neg a \vee b) \wedge (\neg b \vee c) \wedge (a \oplus b \oplus \top) \wedge (b \oplus c \oplus \top)$. The xor-implied literals b^{R_3} and c^{R_4} are in parentheses to indicate that in practice the same literals are not added twice in the trail, but this possibility has to be taken into account when implementing the fully saturated xor-propagation in the integrated CDCL/XOR SAT solver.

4.4 Minimal XOR-Propagation

In this section, we will present the second proposed propagation strategy. As the name *minimal xor-propagation* suggests, the aim is to do only the necessary computation in the xor-reasoning module and resort to unit propagation in the cnf-part whenever possible. In the minimal xor-propagation strategy the following order of preference for the state transition rules is applied: UnitPropagate, XorPropagate, XorImply, Decide. This means that all literals known by the SAT solver are first propagated to the xor-reasoning module before computing any xor-consequences. The minimal xor-propagation search method is presented in Algorithm 6. The differences between minimal xor-propagation and fully saturated xor-propagation are in lines 5-14. Instead of computing xor-consequences in the xor-reasoning module after each propagated literal, all literals are propagated first to the xor-reasoning module (lines 5-6) and after that possibly one xor-consequence is computed (line 7) which is then added to the trail along with a reason (line 13). Only one xor-consequence is computed at a time because a xor-implied literal may be used to infer more literals using unit propagation. That is why unit propagation is performed each time a xor-implied literal is added to the trail of the SAT solver (line 14). As all literals known by the SAT solver are propagated to the xor-reasoning module before computing xor-consequences, the xor-reasoning module cannot return any xor-implied literals known already by the SAT solver (no additional checks in line 13). As the xor-derivations computed in the xor-reasoning module are not fully saturated, the justifying or-clauses returned by XOR-EXPLAIN are not based on first-uip-cuts but on first-cnf-compatible-cuts.

Example 14. *One benefit of the minimal xor-propagation over fully saturated xor-propagation is that less xor-consequences are computed. The trail table in Figure 21 shows a part of the search of the CDCL/XOR SAT Solver on the cnf-xor-formula $\phi = (\neg a \vee b) \wedge (a \oplus c) \wedge (a \oplus d) \wedge (a \oplus e) \wedge (a \oplus b \oplus f \oplus g) \wedge (a \oplus b \oplus f \oplus g \oplus \top)$. The xor-reasoning module derives a conflict from the two xor-assumptions (a) and (b) without computing all xor-consequences of*

Algorithm 6 MINIMAL-XOR-PROPAGATION-SEARCH(ϕ_{or}, ϕ_{xor})

```
 1: π_dpll, π_xor ← (), ()
 2: loop
 3:    (π_dpll, confl) ← UNIT-PROPAGATE(φ_or, π_dpll)
 4:    if not confl then
 5:       for each literal l in π_dpll and not in π_xor do
 6:          π_xor ← XOR-ASSIGN(π_xor, l)
 7:       k ← XOR-DEDUCE(π_xor, φ_xor)
 8:       if k is not undefined then
 9:          C ← XOR-EXPLAIN(k)
10:          if π_dpll ⊨ ¬C then
11:             confl ← C
12:          else
13:             π_dpll ← π_dpll . k^C
14:          goto 2 /* continue with unit propagation */
15:    if confl then
16:       if not HAS-DECISION-LITERALS(π) then
17:          return unsat
18:       (π_dpll, φ_or) ← ANALYZE-CONFLICT(φ_or, confl, π_dpll)
19:       π_xor ← XOR-BACKJUMP(π_dpll)
20:    else
21:       if HAS-UNASSIGNED-VARIABLES(φ_or, π_dpll) then
22:          π_dpll ← ADD-DECISION-LITERAL(φ_or, π_dpll)
23:       else
24:          confl ← XOR-SOLVE(φ_xor, π_xor)
25:          if confl then
26:             goto 15
27:          return sat
```

the xor-assumption (a) which would have to be discarded anyway due to the conflict. For comparison, the same part of the search using fully saturated xor-propagation is shown in Figure 22.

4.5 Postponed XOR-Propagation

If the computational cost of xor-reasoning is substantially higher than unit propagation in the cnf-part, it may be better to consult the xor-reasoning module only when a model for the cnf-part is found. Also, it is possible that literals propagated to the xor-reasoning module can be used to infer a large amount of xor-consequences that are of no use for finding a model for the cnf-part. Computing these unused xor-consequences would just slow the overall search slow. In this section, we will present the third proposed propagation strategy, *postponed xor-propagation*, that postpones the use of the xor-reasoning module until a model for the cnf-part is found.

When a truth assignment that satisfies the cnf-part is found, the trail is propagated one literal at a time to the xor-reasoning module. If the truth assignment is also a model for the xor-part, the search terminates after the whole trail has been propagated to the xor-reasoning module. If the truth

π_{dpll}	π_{xor}	Justification	Rule
1. a^{d}		-	Decide
2. b^{R_1}		$R_1 = \neg a \vee b$	UnitPropagate
	3. a^{d}	(1)	XorPropagate
	4. b^{R_1}	(2)	XorPropagate
		$X = a \oplus b \oplus f \oplus g \oplus \top, \quad C' = \neg a \vee \neg b$	XorConflict

Figure 21: CDCL/XOR SAT solver deduces a conflict using minimal xor-propagation

π_{dpll}	π_{xor}	Justification	Rule
1. a^{d}		-	Decide
2. b^{R_1}		$R_1 = \neg a \vee b$	UnitPropagate
	3. a^{d}	(1)	XorPropagate
4. c^{R_2}		$(a) \wedge (a \oplus c \oplus \top) \models c, \quad R_2 = \neg a \vee c$	XorImply
5. d^{R_3}		$(a) \wedge (a \oplus d \oplus \top) \models d, \quad R_3 = \neg a \vee d$	XorImply
6. e^{R_4}		$(a) \wedge (a \oplus e \oplus \top) \models e, \quad R_4 = \neg a \vee e$	XorImply
	7. b^{R_1}	(2)	XorPropagate
		$X = a \oplus b \oplus f \oplus g \oplus \top, \quad C' = \neg a \vee \neg b$	XorConflict

Figure 22: CDCL/XOR SAT solver deduces a conflict using fully saturated xor-propagation

assignment that satisfies the cnf-part is not a model for the xor-part, a conflict is deduced in the xor-part before the whole trail is propagated to the xor-reasoning module. As the cnf-part is used to drive the search, a part of the SAT solver's trail has to be invalidated using the conflict clause provided by the xor-reasoning module. This makes the solver backjump and try other truth assignments. If the problem instance is not satisfiable, eventually all potential truth assignments are tried and the search terminates giving a negative result.

The xor-reasoning module can be used in this way to check whether a truth assignment is a model for the xor-part of the problem. There is still room for improvement in the use of the xor-reasoning module. When the xor-reasoning module is not used during the search for a model for the cnf-part, the SAT solver has to find a model for the cnf-part without the help of the xor-reasoning module. If the solver had propagated literals to the xor-reasoning module while searching for a model for the cnf-part, the xor-reasoning module could have reduced the number of decision literals needed by propagating xor-implied literals back to the solver. When the SAT solver finds a model for the cnf-part, its trail is likely to contain a number of decision literals. When the trail is propagated to the xor-reasoning module, xor-implied literals can be used to rewrite the trail. Each xor-implied literal is inserted in the trail on its own decision level. The SAT solver performs unit propagation after each modification to the trail. Unit propagation may infer new facts or terminate in a conflict. The purpose of rewriting the trail using xor-implied literals is to eliminate unnecessary decision levels and to cut an

π_{dpll}	π_{xor}	Justification	Rule	
1. a^{d}				Decide
2. b^{R_1}			$R_1 = \neg a \vee b$	UnitPropagate
3. c^{d}				Decide
4. d^{R_2}			$R_2 = \neg c \vee d$	UnitPropagate

Figure 23: Model for the cnf-part found

π_{dpll}	π_{xor}	π_{pending}	Justification	Rule
		1. a^{d} 2. c^{d}		
3. a^{d}			(1)	Decide
4. b^{R_1}			$R_1 = \neg a \vee b$	UnitPropagate
	5. a^{d}		(3)	XorPropagate
	6. b^{R_1}		(4)	XorPropagate
7. $\neg c^{R_2}$			$(a) \wedge (b) \wedge (a \oplus b \oplus c \oplus \top) \models \neg c$ $R_2 = \neg a \vee \neg b \vee \neg c$	XorImply

Figure 24: Postponed xor-propagation contradicts a pending decision

unfruitful trail as early as possible.

An overview of the postponed xor-propagation search method is shown in Algorithm 7. The search proceeds as in any conflict-driven SAT solver until a model for the cnf-part is found (lines 3-12). When a model is found, the decision literals new to the xor-reasoning module are copied to a temporary storage (the "pending" trail) and the actual trail is cut in such a way that it contains only literals that have been propagated to the xor-reasoning module (lines 13-14). While there is a pending decision literal waiting and nothing can be propagated in the cnf-part nor in the xor-part, it is added to the trail (lines 16-20). Unit propagation is performed when a pending decision literal or a xor-implied literal is added to the trail (line 21). While no conflict occurs, the literals not known by the xor-reasoning module are propagated to the xor-reasoning module and xor-implied literals literals are added to the trail (lines 23-32). If a conflict occurs, conflict analysis is performed and literals are removed from the trail until the conflict is resolved, and a new search for a model for the cnf-part starts (line 34-35). If the model for the cnf-part is also a model for the xor-part, all pending decision literals are eventually added to the trail without conflicts and the search terminates (line 17-18). The presented method uses fully saturated xor-propagation strategy when a model for the cnf-part is found, but minimal xor-propagation could be used as well. Both xor-propagation strategies combined with the postponed xor-propagation are evaluated empirically in the next chapter.

Example 15. *Besides reporting that a truth assignment is not a model for the xor-part, the xor-reasoning module can be used to enhance further search by cutting the trail in such a way that its literals can no longer be used to deduce a conflict in the xor-part. A part of the search on the cnf-xor-formula* $\phi = (\neg a \vee b) \wedge (\neg c \vee d) \wedge (d \vee e) \wedge (\neg e \vee c) \wedge (a \oplus b \oplus c) \wedge (d \oplus e)$ *is shown in Figure 23.*

Algorithm 7 POSTPONED-XOR-PROPAGATION-SEARCH(ϕ_{or}, ϕ_{xor})

```
 1: π_dpll, π_xor ← (), ()
 2: loop
 3:    (π_dpll, confl) ← UNIT-PROPAGATE(φ_or, π_dpll)
 4:    if confl then
 5:       if not HAS-DECISION-LITERALS(π) then
 6:          return unsat
 7:       (π_dpll, φ_or) ← ANALYZE-CONFLICT(φ_or, confl, π_dpll)
 8:       π_xor ← XOR-BACKJUMP(π_dpll)
 9:    else
10:       if HAS-UNASSIGNED-VARIABLES(φ_or, π_dpll) then
11:          π_dpll ← ADD-DECISION-LITERAL(φ_or, π_dpll)
12:       else
13:          π_pending ← decision literals in π_dpll and not in π_xor
14:          π_dpll ← π_xor
15:          loop
16:             if cnf- and xor-propagation saturated then
17:                if no unassigned variables then
18:                   confl ← XOR-SOLVE(φ_xor, π_xor)
19:                   if confl then
20:                      goto 4 /* conflict analysis */
21:                   return sat
22:                l ← literal l in π_pending and not in π_dpll
23:                π_dpll ← π_dpll(l^d)
24:             (π_dpll, confl) ← UNIT-PROPAGATE(φ_or, π_dpll)
25:             if not confl then
26:                for each literal p in π_dpll not in π_xor do
27:                   π_xor ← XOR-ASSIGN(π_xor, p)
28:                   (k_1, ..., k_n) ← XOR-DEDUCE(π_xor, φ_xor)
29:                   for each xor-consequence k ∈ (k_1, ..., k_n) do
30:                      C ← XOR-EXPLAIN(k)
31:                      if π_dpll ⊨ ¬C then
32:                         confl ← C
33:                         break
34:                      else
35:                         π_dpll ← π_dpll . k^C if k ∉ π_dpll
36:             if confl then
37:                goto 4 /* conflict analysis */
```

The SAT solver picks two decision literals (a) and (c). The literals b and d are inferred by unit propagation. The resulting truth assignment $\{ a, b, c, d \}$ is a model for the CNF part $\{ (\neg a \vee b), (\neg c \vee d) \}$. The horizontal lines in the trail table delimit decision levels. Figure 24 illustrates the process of rewriting the trail using the xor-reasoning module. The trail table is extended with another column π_{pending} to indicate the pending decision literals. The literals (a) and (c) are in π_{pending} and the inferred literals (b) and (d) are left out as they can be inferred again from (a) and (c). The pending decision literals are picked one by one and added to the trail. Unit propagation is

π_{dpll}	π_{xor}	π_{pending}	Justification	Rule
		1. a^{d} 2. c^{d} 3. e^{d}		
4. a^{d}			(1)	Decide
5. b^{R_1}			$R_1 = \neg a \vee b$	UnitPropagate
	6. a^{d}		(3)	XorPropagate
	7. b^{R_1}		(4)	XorPropagate
8. c^{R_2}			$(a) \wedge (b) \wedge (a \oplus b \oplus c) \models c$ $R_2 = \neg a \vee \neg b \vee c$	XorImply
9. d^{R_3}			$R_3 = \neg c \vee d$	UnitPropagate
	10. d^{R_3}		(9)	XorPropagate
11. e^{d}			(3)	Decide
	12. e^{d}		(11)	XorPropagate
			$X{=}d{\oplus}e{\oplus}f{\oplus}g{\oplus}\top$, $C{=}\neg d{\vee}\neg e$	XorConflict

Figure 25: Decision level eliminated by postponed xor-propagation

performed each time after a pending decision literal is added to ensure that the trail is saturated with respect to unit propagation and to detect conflicts. The literals (a) and (b) are propagated to the xor-reasoning module and the xor-reasoning module returns the xor-implied literal $(\neg c)$. The original trail had the decision literal (c) so the trail rewrite contradicts a pending decision. The postponed xor-propagation ends here and a new search for a model for the cnf-part is started.

Example 16. *Xor-implied literals can also be used to eliminate decision levels. When a model for the cnf-part is found, the trail is propagated to the xor-reasoning module and it is rewritten in such a way that the justifying or-clause of a xor-implied literal can used to infer the originally heuristically picked decision literal. After this the search can arguably continue more effectively. Figure 25 shows how the trail is rewritten and one of pending decision literals is eliminated. The cnf-xor-formula in question is $\phi = (\neg a \vee b) \wedge (\neg c \vee d) \wedge (a \oplus b \oplus c) \wedge (d \oplus e \oplus f \oplus g) \wedge (d \oplus e \oplus f \oplus g \oplus \top)$. The pending decision literal (a) is put back to the trail and the literal (b) is inferred by unit propagation. The literals (a) and (b) are then propagated to the xor-reasoning module. The xor-reasoning module returns the xor-implied literal (c). The literal (c) is added to the trail. The literal (d) is inferred by unit propagation and propagated to the xor-reasoning module. As the literal (c) was returned as a xor-implied literal, a decision level is eliminated from the trail. The literal (e) causes a xor conflict when propagated to the xor-reasoning module and the postponed xor-propagation ends there.*

Example 17. *Even if decision levels cannot be eliminated, trail rewrite can still move literals to earlier decision levels in trail. Intuitively, the less assumptions are needed to infer a literal, the more useful the literal is in finding a model for the cnf-xor-formula. The trail table in Figure 26 shows how*

π_{dpll}	π_{xor}	Justification	Rule
1. a^{d}			Decide
2. b^{R_1}		$R_1 = \neg a \vee b$	UnitPropagate
3. c^{d}			Decide
4. d^{R_2}		$R_2 = \neg c \vee d$	UnitPropagate
5. e^{R_3}		$R_3 = \neg c \vee \neg d \vee e$	UnitPropagate

Figure 26: Model for cnf-part found

π_{dpll}	π_{xor}	π_{pending}	Justification	Rule
		1. a^{d} 2. c^{d}		
3. a^{d}			(1)	Decide
4. b^{R_1}			$R_1 = \neg a \vee b$	UnitPropagate
	5. a^{d}		(3)	XorPropagate
	6. b^{R_1}		(4)	XorPropagate
7. e^{R_2}			$(a) \wedge (b) \wedge (a \oplus b \oplus e) \models e$ $C = \neg a \vee \neg b \vee e$	XorImply
8. c^{d}			(2)	Decide
9. d^{R_3}			$R_3 = \neg c \vee d$	UnitPropagate
	10. c^{d}		(8)	XorPropagate
	11. d^{R_3}		(9)	XorPropagate
			$X = a \oplus b \oplus c \oplus d$ $C = \neg a \vee \neg b \vee \neg c \vee \neg d$	XorConflict

Figure 27: Literal moved to an earlier decision level by postponed xor-propagation

the SAT solver has found a model for the cnf part of the cnf-xor-formula $\phi = (\neg a \vee b) \wedge (\neg c \vee d) \wedge (\neg c \vee \neg d \vee e) \wedge (a \oplus b \oplus e) \wedge (a \oplus b \oplus c \oplus d) \wedge (c \oplus d)$ *In the trail table in Figure 27 the literals* (a) *and* (b) *are propagated to the xor-reasoning module which propagates back the xor-implied literal* (e)*. The second pending decision literal* (c) *and the inferred literal* (d) *are propagated next to the xor-reasoning module. This enables the xor-reasoning module to deduce a conflict in the xor-part. Due to the conflict clause* $\neg a \vee \neg b \vee \neg c \vee \neg d$*, the decision* (c) *has to be undone and the literal* (e) *can no longer be inferred with the clauses* $\neg c \vee d$ *and* $\neg c \vee \neg d \vee e$*. However, the or-clause* $\neg a \vee \neg b \vee e$ *that justifies the xor-implied literal* (e) *enables the solver to infer the literal* (e) *and on an earlier decision level than before.*

The postponed xor-propagation strategy is similar to lazy SMT theory propagation of the DPLL(T) framework (see Section 3.2.2 in [28]) with one addition: the trail of the SAT solver is rewritten using xor-implied literals as described in this section.

4.6 Handling XOR-Implied Literals in SAT Solver

In this section, we present alternative ways for handling the justifying or-clauses of xor-implied literals and the conflict clauses returned by the xor-reasoning module in the SAT solver. Upon deducing a conflict, the SAT solver builds a conflict clause. The conflict clause is then added to the database of learned or-clauses. In order to avoid memory congestion and computational overhead caused by more expensive unit propagation, the number of learned or-clauses is kept bounded by an upper limit. The maximum number of learned or-clauses is increased periodically to ensure that the search terminates. When the xor-reasoning module is integrated to the SAT solver, the justified or-clauses and the conflict clauses provided by the xor-reasoning module can either be stored or they can be used directly in the conflict analysis of the SAT solver. We will present three strategies for handling the or-clauses returned by the xor-reasoning module in the SAT solver.

1. Storing nothing. If storing or-clauses involves a considerable cost, it may be more efficient to use the or-clauses provided by the xor-reasoning module directly in the conflict analysis and discard them afterwards. When analyzing a conflict, the SAT solver needs to produce an asserting conflict clause which causes the SAT solver to undo at least one of the decision literals. An efficient learning scheme for computing conflict clauses is presented in the paper by Zhang et al. [38]. In order to produce an asserting conflict clause, the SAT solver has to track which decision literals caused the conflicting or-clause. The conflicting clause may contain literals that are not decision literals. Reasons for these non-decision literals have to be tracked by inspecting the or-clauses that were used to infer the non-decision literals by unit propagation. The or-clauses inferring the non-decision literals may again contain more non-decision literals and the process of finding the decision literals involved in the conflict continues recursively. Conflict clauses may also contain non-decision literals, but the details of conflict analysis methods of a SAT solver are outside the scope of this report. For our purposes, it suffices to know that the SAT solver needs an or-clause that justifies each non-decision literal

in order to perform conflict analysis. When literals of the trail of the SAT solver are propagated to the xor-reasoning module, the xor-reasoning module may return a number of xor-implied literals. If xor-implied literals are used to infer a conflicting or-clause, the SAT solver needs a justifying or-clause for each xor-implied literal involved in the conflict. The conflict analysis method of the SAT solver can be modified in such a way that xor-implied literals are tagged so that the SAT solver can ask the xor-reasoning module to return a justifying or-clause when one is needed (using the method EXPLAIN). The SAT solver's database of learned clauses will only contain the resulting conflict clauses so the bounded number of slots for learned clauses is used effectively. Also, unit propagation works faster when there are less or-clauses to check. As a downside, possibly more justifying or-clauses are computed when they are not stored.

2. Storing all or-clauses. This strategy is based on the assumption that justifying or-clauses of xor-implied literals may capture useful pieces of the xor-part which, when combined with the cnf-part, may prune large parts of the search space. It is also assumed that the benefit of storing justifying or-clauses outweighs the cost of computing them. Whenever a xor-implied literal is returned by the xor-reasoning module, a justifying or-clause is computed for it and added to the SAT solver's database of learned clauses. If a xor-implied literal is needed in conflict analysis, its justifying or-clause is already available. When the xor-reasoning module operates effectively, the number of xor-implied literals is high. This means that the number of stored or-clauses gets high as well. The maximum number of stored or-clauses is bounded so when the limit is reached, one of the unused or-clauses has to be removed. For efficiency reasons, this can be done by removing a number of least active or-clauses in batches. If the database of learned clauses is constantly flooded by justifying or-clauses for xor-implied literals, it is possibly that potentially useful or-clauses get removed. Also, keeping the number of learned clauses near the allowed maximum may cause unit propagation to perform slower.

3. Storing useful or-clauses. If all justifying or-clauses for xor-implied literals are stored in the database of learned clauses, this eventually performs translation of the xor-part to CNF. However, if no justifying or-clauses are stored, more computation is done in the xor-reasoning module. The third proposed strategy for handling justifying or-clauses for xor-implied literals is a compromise between the two strategies presented above: justifying or-clauses for xor-implied literals that participate in conflicts are stored. Considering the satisfiability of a xor-clause X, there are 2^{n-1} different truth assignments involving variables of X that are not models for X. This implies that there are many ways to derive a conflicting xor-clause. Due to the symmetry of xor-clauses, xor-refutations from different xor-assumptions may include parts that are identical and have the same xor-implied literals. That is why we consider storing justifying or-clauses for xor-implied literals that participate in conflicts potentially advantageous. The preliminary performance tests suggested that the third strategy is likely to perform well, so it is used in the empirical evaluation in the next chapter.

5 EXPERIMENTAL RESULTS

In this chapter, we present results of the experimental evaluation of our proof-of-concept implementation and compare it to other SAT solvers that are designed with xor-clauses in mind. We have evaluated the efficiency of our approach by integrating the xor-reasoning module to minisat [12] (version 2.0 core), an efficient yet simple conflict-driven solver. We call the version of minisat enhanced with xor-reasoning xor-minisat. We considered problem 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 benchmarks are : known-plaintext attack (see e.g. Chapter 2 in [35]) on the block cipher DES [1], random generated linear problems based on 3-regular bipartite graphs [13], and a known keystream attack on the stream cipher Trivium [9]. Due to the heuristic nature of operation of SAT solvers, variance in solving times can be very significant when solving similar instances. This is why we generated a number of instances from each benchmark. Different search methods can then be compared based on the average performance with respect to two characteristics : solving time and number of heuristic decisions needed to complete the search. All tests were run on a Six-Core AMD Opteron^{TM} with the CPU frequency of 2.60GHz and 512 KB of cache memory running Linux as the operating system. The tests had a hard memory limit of 2GB. A test was terminated if it did not stop executing after four hours.

The attacks on the cryptographic ciphers were generated by first modelling a Boolean circuit of the execution of the cipher leaving the input gates corresponding to the key bits unspecified and then converting the Boolean circuit into a cnf-xor-formula in a DIMACS-like format where constraints can be expressed as xor-clauses, too. We generated also CNF-only versions of the benchmarks for comparison. The CNF-only version was generated from the cnf-xor-formula by substituting each xor-clause with a logically equivalent conjunction of or-clauses without introducing new variables.

In addition to comparing xor-minisat to minisat, we included the following solvers which employ some kind of equivalence-/xor-reasoning in the empirical study : cryptominisat, EqSatz, march_eq, moRsat, and 2clseq.

In order to evaluate different xor-propagation strategies and alternative ways for reasoning on the xor-part, we compare 16 variants of our solver xor-minisat. A variant xor-minisat^{ABCD} is identified by a four-letter superscript $ABCD$ where:

- A tells the xor-propagation strategy
  - $A = e$ if the xor-propagation strategy is fully saturated or minimal
  - $A = p$ if the xor-propagation strategy is postponed.
- B tells how the xor inference rules are prioritized
  - $B = m$ if the (inner) xor-propagation strategy is minimal

- $B = f$ if the (inner) xor-propagation strategy is fully saturated

- C tells if xor-internal variables are returned in reason sets
  - $C = i$ if the xor-internal variables are not returned in reason sets computed by the xor-reasoning module
  - $C = s$ if all variables are treated as xor-shared

- D tells whether the reason sets are minimized
  - $D = t$ if the reason sets are minimized
  - $D = f$ if the reason sets are not minimized

The xor-minisat-variant xor-minisat^{emst} solved most problems of all variants so we compare it to other solvers.

5.1 Block Cipher DES

In this section, we present results of the evaluation of a known-plaintext attack on a reduced yet challenging configuration of the block cipher DES : 4 rounds with 1 block. Only 1% of the constraints are xor-clauses in these instances so we did not expect that xor-reasoning would contribute much to the search. 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). This benchmark is included in order to show that the xor-reasoning module does not incur an unbearable computational overhead.

Table 28 contains a comparison of all solvers and xor-minisat-variants. The solver xor-minisat^{efsf} performs best on average of the variants so we included it in comparison in the following figures.

Figure 29 illustrates the number of solved DES instances (4 rounds, 1 blocks) with respect to the solving time. Comparing the variants of xor-minisat to the unmodified version minisat indicates that xor-minisat^{emst} has a constant overhead in solving time while xor-minisat^{efsf} solves majority of the instances comparably to minisat. The solver cryptominisat performs comparably to the solver xor-minisat^{emst}. The solver moRsat is the slowest solver to solve all DES instances. The solvers march_eq, 2clseq, EqSatz, and lsat do not solve any instances of the DES benchmark. Figure 30 shows the number of solved DES instances as a function of heuristic decisions needed. The solver xor-minisat^{emst} using minimal xor-propagation strategy requires more heuristic decisions to solve the DES instances than minisat while the solver xor-minisat^{efsf} using fully saturated xor-propagation requires less heuristic decisions than minisat. Unit propagation can be performed in the similar way in the xor-part as in the cnf-part so we suspect that the increase in the number of heuristic decisions (with xor-minisat^{emst}) may be due to how the xor-reasoning module computes reason sets for xor-implied literals (and conflicts). The way how reason sets are computed may affect decision heuristics of the SAT solver and increase the number of heuristic decisions. The solver xor-minisat^{emst} computes reason sets based on first cnf-compatible cuts

Solver	# Solved	Time median (s)	Decisions median
2clseq	0	-	-
cryptominisat	32	1460.18	873556
EqSatz	0	-	-
march_eq	0	-	-
minisat	32	377.34	948662
moRsat	32	1391.42	4571775
lsat	0	-	-
xor-minisat^{emif}	32	669.81	1455312
xor-minisat^{emit}	32	753.04	1455312
xor-minisat^{efsf}	32	515.86	890447
xor-minisat^{efst}	32	631.50	890447
xor-minisat^{efif}	32	598.52	920702
xor-minisat^{efit}	32	653.40	920702
xor-minisat^{emsf}	32	782.22	1579648
xor-minisat^{emst}	32	918.82	1579648
xor-minisat^{pmif}	32	806.31	1867684
xor-minisat^{pmit}	32	978.46	1867684
xor-minisat^{pfsf}	32	567.31	1225031
xor-minisat^{pfst}	32	523.84	1225031
xor-minisat^{pfif}	32	541.43	1213144
xor-minisat^{pfit}	32	593.07	1213144
xor-minisat^{pmsf}	32	1149.86	2451015
xor-minisat^{pmst}	32	1183.66	2451015

Figure 28: DES benchmark results (32 instances, 4 rounds with 1 block)

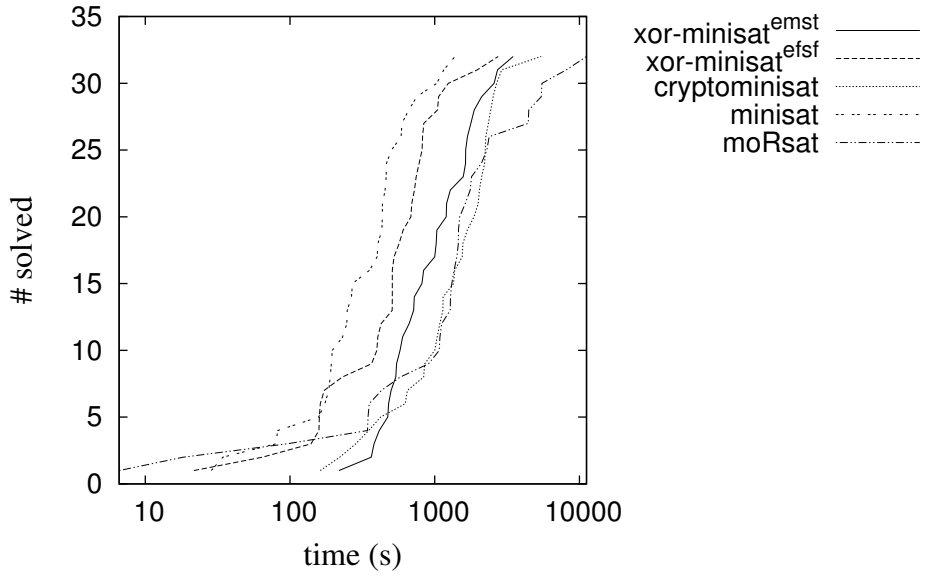


Figure 29: Number of solved DES instances w.r.t time

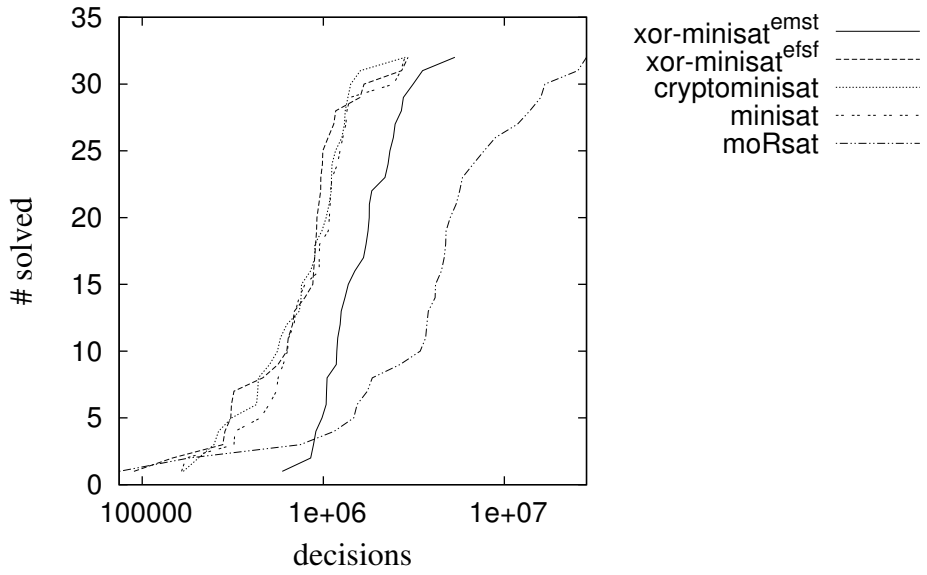


Figure 30: Number of solved DES instances w.r.t decisions

and the solver xor-minisat^{efsf} computes reason sets based on cnf-compatible first-UIP-cuts. The results agree with our intuition that the postponed xor-propagation (the solver xor-minisat^{pfsf}) is effective on instances where the cnf-part dominates the search. The number of heuristic decisions is higher when using postponed xor-propagation because less propagation can be done when a model for the cnf-part is searched. Despite the higher number of heuristic decisions, the decisions are computationally cheaper because xor-propagation is not performed before a model for the cnf-part is found and thus, the number of heuristic decisions is not directly comparable to other xor-propagation strategies. There was only one xor-internal variable in each instance so it is reasonable that whether xor-internal variables are returned to the SAT solver or not has little significance. Fully saturated xor-propagation performs clearly better than minimal xor-propagation based on the comparison of solving time medians in Figure 28.

5.2 Randomly Generated Linear Problems

In this section, we discuss the results of the second benchmark: regular XOR-SAT [13] – artificial problem instances consisting only of xor-clauses based on random generated 3-regular bipartite graphs. As the instances of regular XORSAT are linear, they can effectively be solved as-such by Gaussian elimination, so when solved by variants of xor-minisat, the instances were modified by selecting xor-clauses randomly and converting them to CNF until a specified proportion of xor-clauses is converted to CNF. The problem instances included satisfiable and unsatisfiable instances with a number of variables ranging from 96 to 240.

The representation of the constraints - whether they are expressed as xor-clauses or as or-clauses - has an effect on the time spent in solving a reg-

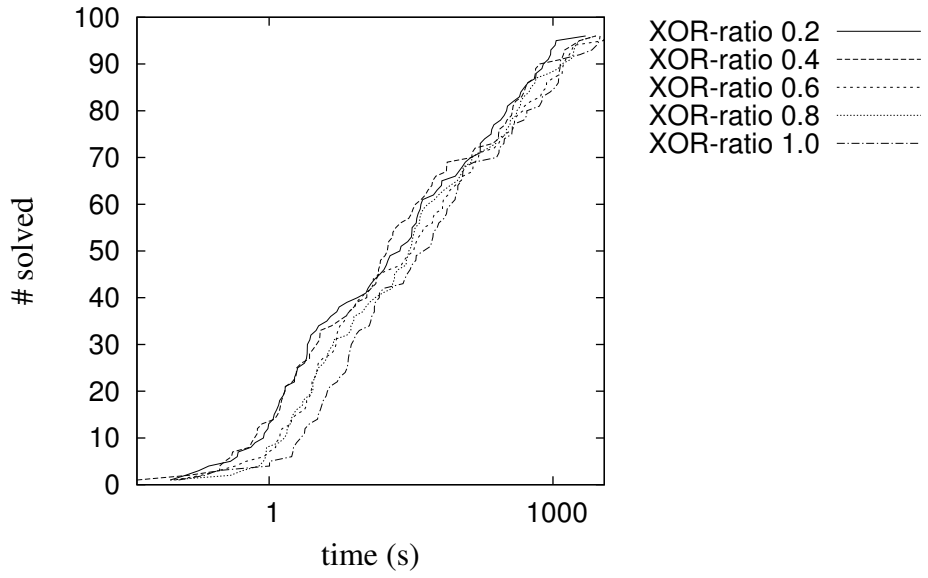


Figure 31: Number of solved satisfiable regular XORSAT instances w.r.t time (different XOR-ratios, e.g the XOR-ratio 0.2 meaning that 80% of the xor-clauses are converted to CNF) by xor-minisat^{emst}.

ular XORSAT instance, especially with unsatisfiable instances. Figure 31 shows how the time needed in solving an instances grows by a constant factor when the XOR-ratio, the proportion of xor-clauses of all constraints, is increased. The tests were solved by xor-minisat^{emst}. Figure 32 shows that the number of heuristic decisions needed to solve an instance is smallest when all constraints are expressed as xor-clauses (XOR-ratio 1.0). However, using the current implementation techniques, the xor-reasoning seems to be computationally more expensive than unit propagation in the cnf-part so the smaller the XOR-ratio is, the shorter the expected runtime of solving an instance is. When a regular XORSAT instance is satisfiable, the use of the xor-reasoning module reduces the number of heuristic decisions needed. With unsatisfiable regular XORSAT instances, the situation is different. The more the constraints are expressed as xor-clauses, the harder it becomes for the SAT solver to prove that an instance is unsatisfiable. This is shown in Figure 34. The higher number of heuristic decisions needed is reflected in longer solving time in Figure 33. This may be caused by how the representation of constraints decisions heuristics of the SAT solver. The smaller the cnf-part is, the less information the SAT solver has to guide its decision heuristics.

We decided to use the XOR-ratio 0.4 for comparing xor-minisat to other solvers because the xor-reasoning module is used to operate on a substantial part of constraints and still performs reasonably well. Figure 35 shows how other solvers and other xor-minisat-variants solved satisfiable regular XORSAT instances. In Figure 36 there is a similar comparison but for unsatisfiable regular XORSAT instances. The solvers 2clseq, cryptominisat, march_eq, and moRsat did not perform comparably in this benchmark. The solver EqSatz solved satisfiable regular XORSAT instances fast and with a small number of

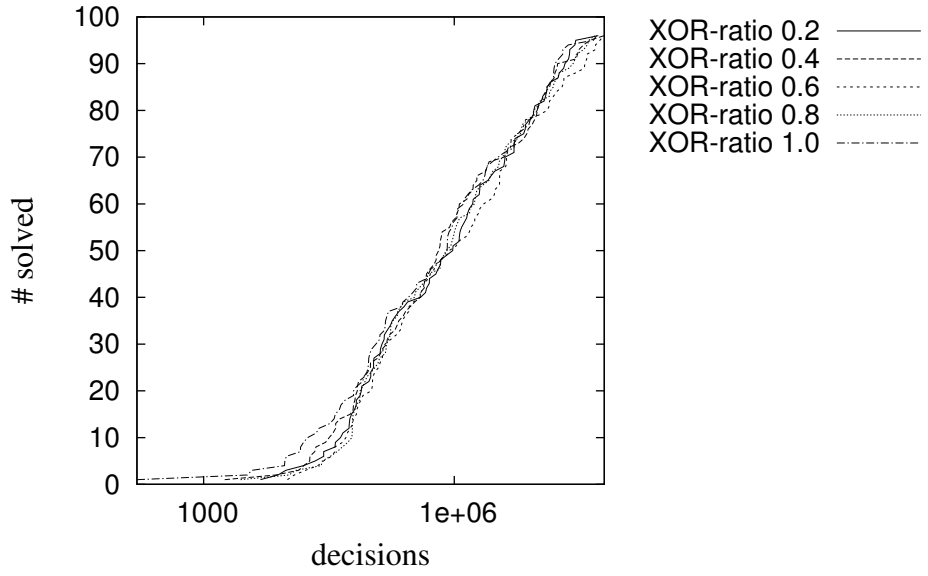


Figure 32: Number of solved satisfiable regular XORSAT instances w.r.t decisions (different XOR-ratios) by xor-minisat^{emst}

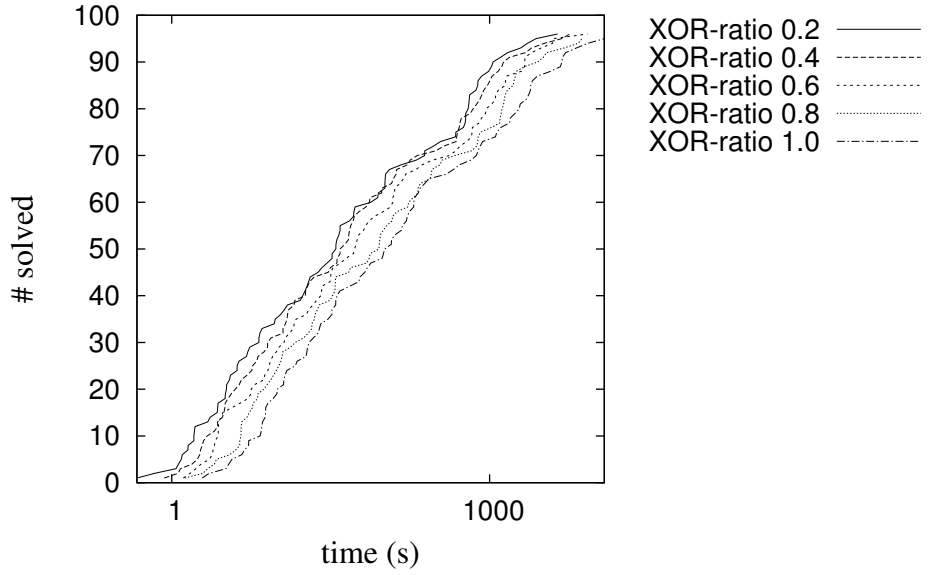


Figure 33: Number of solved unsatisfiable regular XORSAT instances w.r.t time (different XOR-ratios) by xor-minisat^{emst}

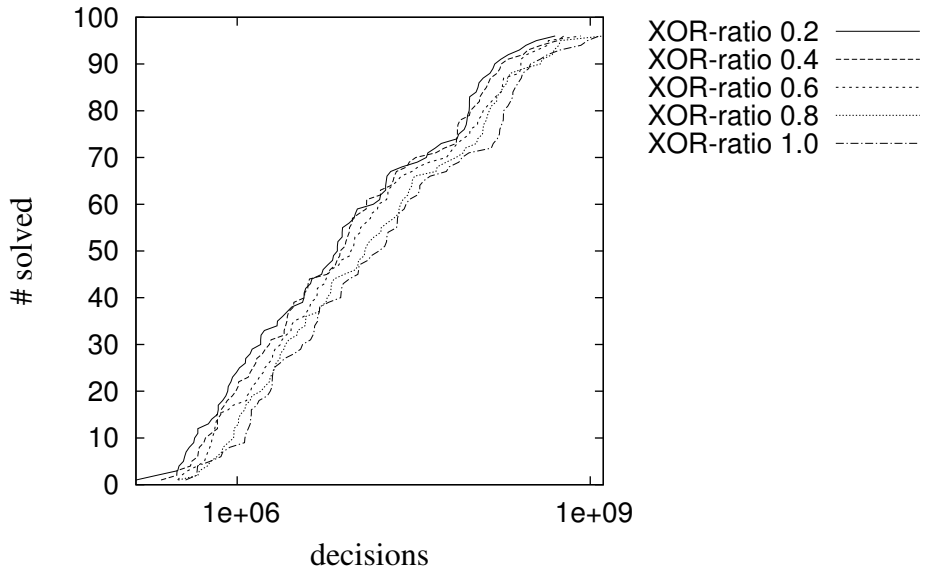


Figure 34: Number of solved unsatisfiable regular XORSAT instances w.r.t time (different XOR-ratios) by xor-minisat^{emst}

heuristic decisions, but performed poorly with unsatisfiable instances. The solver lsat solved both unsatisfiable and satisfiable regular XORSAT instances almost instantly. If xor-internal variables are not returned to the SAT solver, this adds a significant overhead to both solving time and the number of heuristic decisions needed when solving unsatisfiable regular XORSAT instances and the performance decreases when solving satisfiable instances, too. The number of solved regular XORSAT instances with respect to the solving time are shown in Figures 37 (satisfiable instances) and 39 (unsatisfiable). The number of heuristic decisions needed to solve regular XORSAT instances are shown in Figures 38 and 40. The unmodified minisat still performs better with both satisfiable and unsatisfiable regular XORSAT instances than any of the xor-minisat-variants. The use of the xor-reasoning module does not reduce the number of heuristic decisions needed but does not increase it either. This may be due to the highly regular and delinearized structure of XORSAT instances.

5.3 Stream cipher Trivium

In this section, we present the results of the experimental of the third benchmark: a "known key-stream attack" on stream cipher Trivium modelled by generating a small number of keystream bits (from one to twenty in the benchmark) after 1152 initialization rounds. The 80-bit initial value vector is randomly generated and given in the problem instance. The 80-bit key is left open. As the generated keystream bits are far fewer than the key bits, a number of keys produce the same prefix with a high probability. The structure of the Boolean circuit generated from Trivium suggests that xor-reasoning may be beneficial. All Trivium instances contained either two or

Solver	# Solved	Time median (s)	Decisions median
2clseq	37	-	-
cryptominisat	7	-	-
EqSatz	96	0.85	3451
lsat	96	0.00	0
march_eq	0	-	-
minisat	96	10.06	549750
moRsat	0	-	-
xor-minisat^{emif}	96	17.08	624424
xor-minisat^{emit}	96	17.12	624424
xor-minisat^{efsf}	96	19.40	721610
xor-minisat^{efst}	96	19.40	721610
xor-minisat^{efif}	96	32.10	1162048
xor-minisat^{efit}	96	32.13	1162048
xor-minisat^{emsf}	96	15.97	609030
xor-minisat^{emst}	96	15.93	609030
xor-minisat^{pmif}	90	25.68	1433932
xor-minisat^{pmit}	92	26.84	1525869
xor-minisat^{pfsf}	79	33.14	1779888
xor-minisat^{pfst}	79	33.20	1779888
xor-minisat^{pfif}	89	26.80	1502614
xor-minisat^{pfit}	89	26.76	1502614
xor-minisat^{pmsf}	84	24.56	1353987
xor-minisat^{pmst}	84	24.17	1294190

Figure 35: Regular XORSAT benchmark results (satisfiable, 96 instances)

Solver	# Solved	Time median (s)	Decisions median
2clseq	0	-	-
cryptominisat	15	-	-
EqSatz	39	-	-
lsat	96	0.00	0
march_eq	0	-	-
minisat	96	24.71	6887774
moRsat	0	-	-
xor-minisat^{emif}	91	144.92	24605620
xor-minisat^{emit}	91	145.24	24605620
xor-minisat^{efsf}	96	34.55	6869888
xor-minisat^{efst}	96	34.59	6869888
xor-minisat^{efif}	91	127.06	21463326
xor-minisat^{efit}	91	126.84	21463326
xor-minisat^{emsf}	96	37.60	7253954
xor-minisat^{emst}	96	37.61	7253954
xor-minisat^{pmif}	90	146.75	38852266
xor-minisat^{pmit}	90	146.70	38852266
xor-minisat^{pfsf}	96	31.59	9433134
xor-minisat^{pfst}	96	31.61	9433134
xor-minisat^{pfif}	90	142.97	40543844
xor-minisat^{lfit}	90	143.09	40543844
xor-minisat^{pmsf}	96	29.90	8808420
xor-minisat^{pmst}	96	30.03	8808420

Figure 36: Regular XORSAT benchmark results (unsatisfiable, 96 instances)

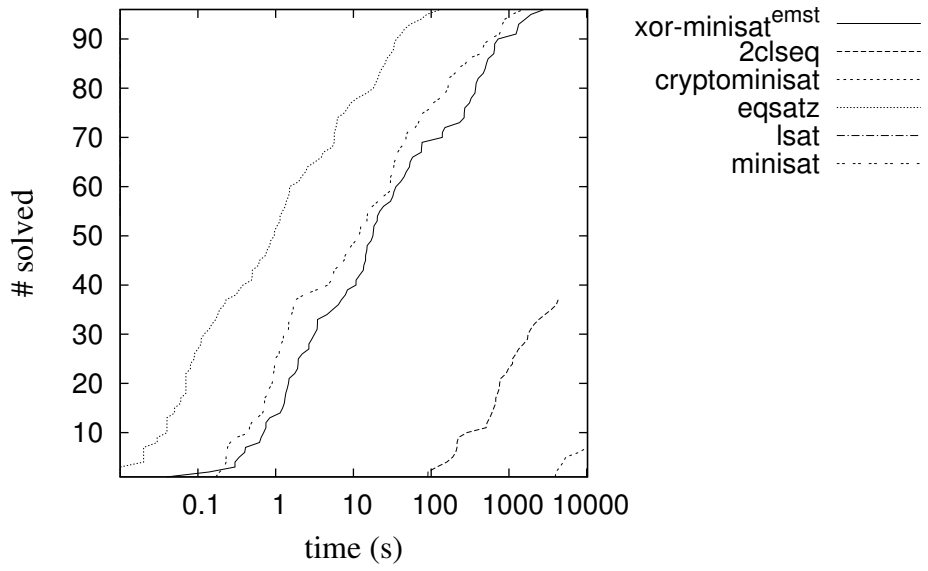


Figure 37: Number of solved satisfiable regular XORSAT instances w.r.t time

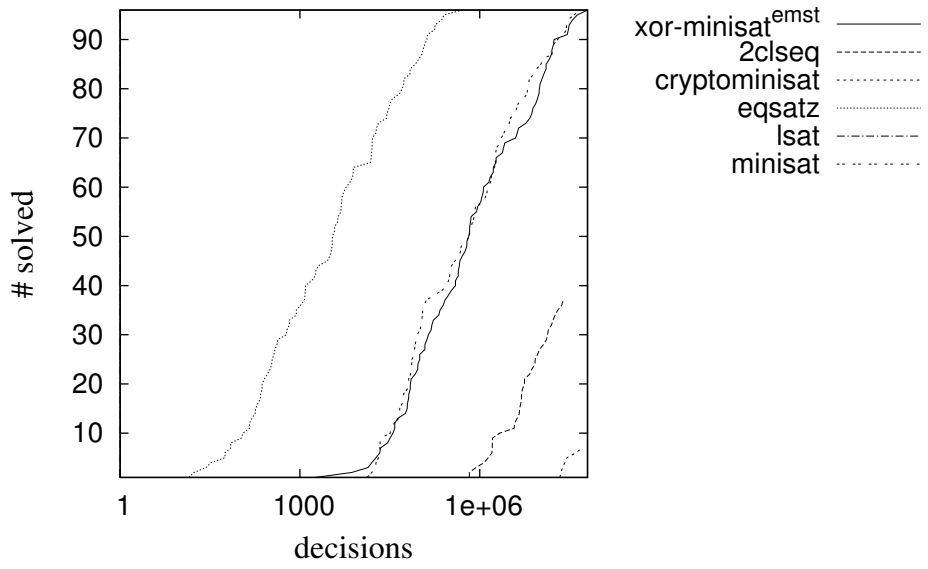


Figure 38: Number of solved satisfiable regular XORSAT instances w.r.t decisions

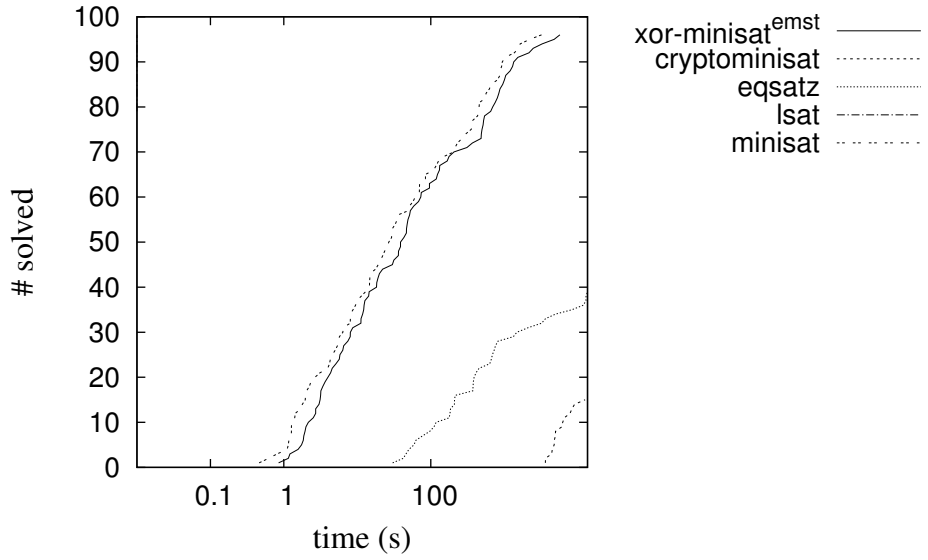


Figure 39: Number of solved unsatisfiable regular XORSAT instances w.r.t time

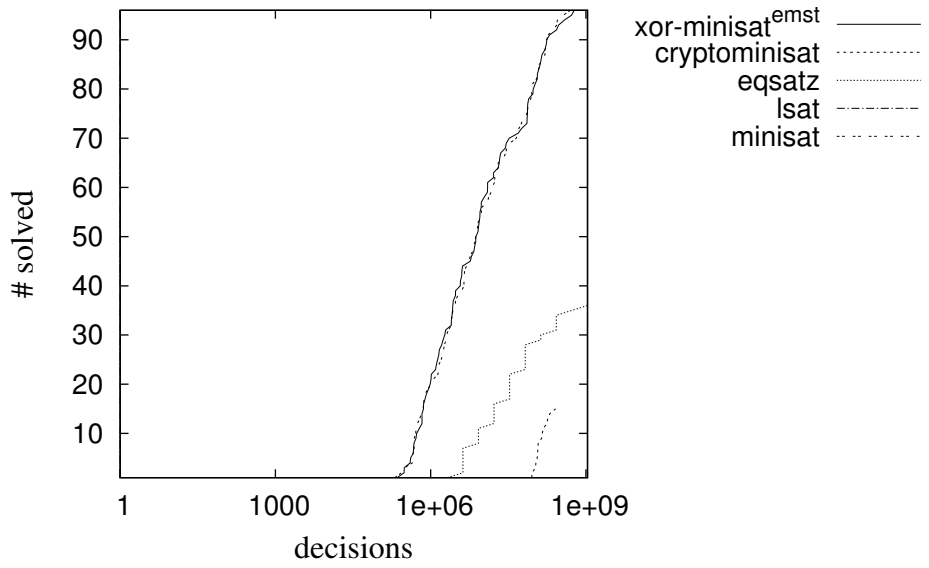


Figure 40: Number of solved unsatisfiable regular XORSAT instances w.r.t decisions

three large clusters of xor-clauses. For instance, if the number of clusters is three, each cluster had 2600–2900 xor-clauses involving 3500–3800 variables. The cnf-part contains typically 8000–8600 or-clauses involving 5250–5700 variables. In Figure 41 there is a comparison of the solvers and xor-minisat-variants. The solvers moRsat, march_eq and minisat perform best in the comparison. The solvers 2clseq, EqSatz, lsat, and cryptominisat and all xor-minisat-variants using postponed xor-propagation manage to solve fewer instances. The number of solved Trivium instances with respect to solving time is shown in Figure 42. Figure 43 shows the number of heuristic decisions needed to solve Trivium instances with different solvers. It seems that the hybrid approach of combining conflict-driven clause learning with look-ahead decision heuristics employed by moRsat is effective in solving Trivium instances. The preprocessing of the xor-part and look-ahead techniques done by march_eq seem to work well in Trivium benchmark, too. In most cases, minimizing reason sets does not have any effect on the length of the search. However, some of the Trivium instances are big enough to demonstrate that removing redundant literals from reason sets does make a difference. As the xor-part is overwhelming in Trivium instances, postponed xor-propagation performs poorly as expected. Minimal xor-propagation combined with hiding xor-internal variables from the SAT solver works best with Trivium instances. The unmodified version of minisat is faster with small instances, but as bigger instances are solved, xor-minisat starts to perform better than regular minisat.

Solver	# Solved	Time median (s)	Decisions median
2clseq	101	-	-
cryptominisat	554	25.69	10594
EqSatz	546	114.70	258
lsat	25	-	-
march_eq	627	5.81	479
minisat	579	6.76	10290
moRsat	639	2.63	4108
xor-minisat^{emif}	607	23.52	2198
xor-minisat^{emit}	610	23.09	2142
xor-minisat^{efsf}	594	59.12	3977
xor-minisat^{efst}	597	60.55	3977
xor-minisat^{efif}	587	42.67	3197
xor-minisat^{efit}	589	42.83	3197
xor-minisat^{emsf}	600	36.09	2905
xor-minisat^{emst}	608	27.06	2184
xor-minisat^{pmif}	343	57.31	3643179
xor-minisat^{pmit}	342	66.43	4005323
xor-minisat^{pfsf}	387	108.93	23756148
xor-minisat^{pfst}	387	111.76	23756148
xor-minisat^{pfif}	351	106.77	3688900
xor-minisat^{lfit}	351	105.23	3688900
xor-minisat^{pmsf}	392	94.51	22494246
xor-minisat^{pmst}	390	95.10	22768119

Figure 41: Trivium benchmark results (640 instances)

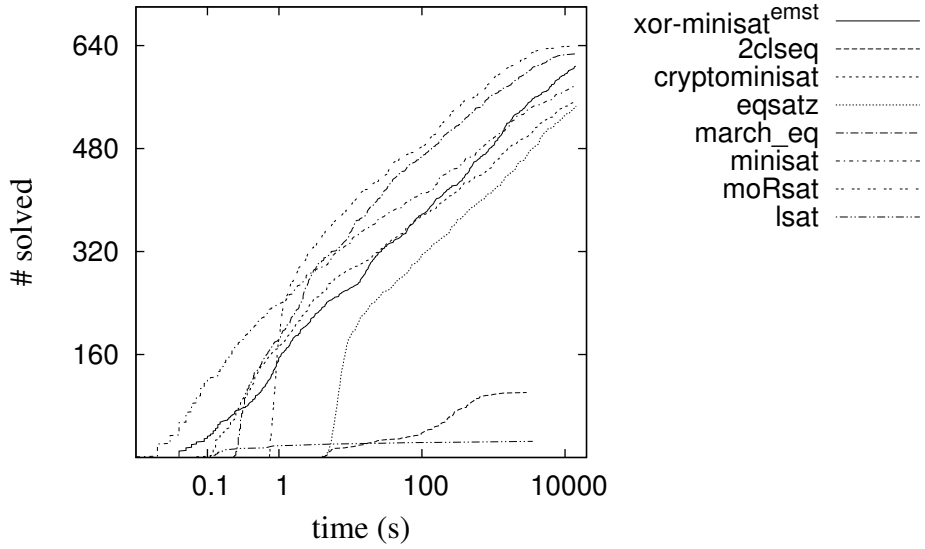


Figure 42: Number of solved Trivium instances w.r.t time

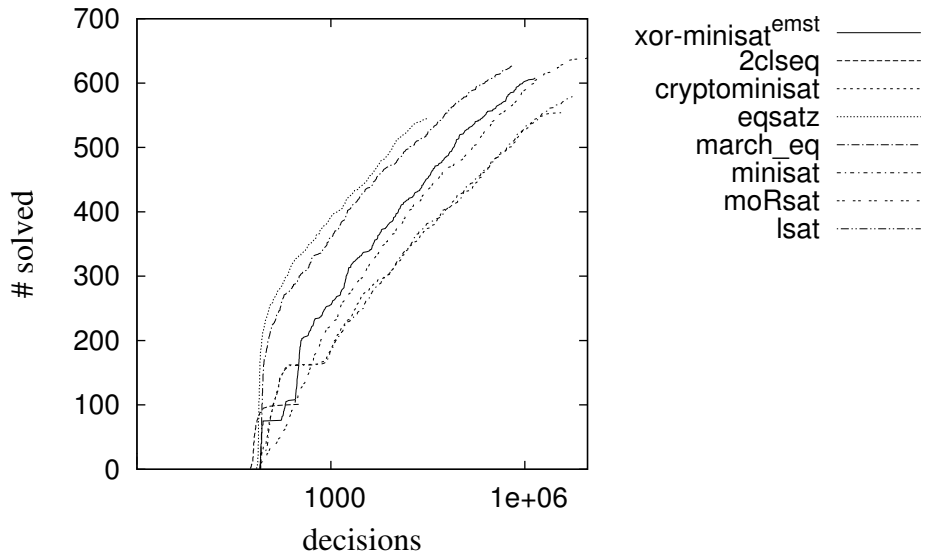


Figure 43: Number of solved Trivium instances w.r.t decisions

6 CONCLUSIONS

The research problem studied in this report is to find an effective method for solving an important class of the propositional satisfiability problem: to decide whether the variables of a propositional formula consisting of a conjunction of or-clauses and xor-clauses can be assigned in such a way that the formula evaluates to "true". Such problems, arising in application domains such as logical cryptanalysis and circuit verification, make modern SAT solvers scale poorly due to parity constraints (xor-clauses) occurring in problem descriptions.

The research problem is addressed by a novel approach taking advantage of the strengths of continuously improving SAT technology through the use of a solver module that decides satisfiability of a conjunction of xor-clauses. Such a solver module is integrateable to existing and future conflict-driven clause learning SAT solvers through a minimal interface according the DPLL(T) framework. In order to support conflict-driven clause learning, the solver module has to be able to explain how truth values for variables are deduced. The use of DPLL-based search methods suggests that computational operations of the solver module should be carried out in an incremental and backtrackable fashion. In order to benefit from separating parity constraints from CNF, the proof system of the solver module has to be stronger than unit propagation in CNF. Due to extremely well fine-tuned algorithms and data structures of modern SAT solvers, the solver module has to be fast in order to enhance the performance of the SAT solver.

This report devises a xor-reasoning module for reasoning about satisfiability of a conjunction of xor-clauses and develops alternatives for integrating the xor-reasoning module loosely according to the DPLL(T) framework into a modern conflict-driven clause learning SAT solver minisat, resulting

in a hybrid solver xor-minisat. Due to the minimal interface presented, the xor-reasoning module is likely to be easily integrated to many SAT solvers with reasonable effort. The xor-reasoning module enhances the search by inferring truth values for variables and explaining how the truth values are deduced when necessary. The inference rules of the xor-reasoning module and strategies for handling variables that occur only in the xor-clauses of the problem instances are presented. Alternative ways for computing clausal explanations for conflicts and a method for minimizing such explanations are introduced. Various alternatives for integrating the xor-reasoning module to a SAT solver are discussed ranging from computing all computable logical consequences in the xor-reasoning module whenever possible to postponing the use of the xor-reasoning module until a model for the cnf-part of the problem instance is found. Possibilities for using clausal explanations provided by the xor-reasoning module in the SAT solver are presented.

We have developed a prototype implementation of the xor-reasoning module and integrated it into minisat, a simple yet efficient conflict-driven SAT solver. Computational operations of the xor-reasoning module are performed incrementally. The state of the xor-reasoning module can be stored and restored without significant computational overhead and unbearable increase in the use of memory. We evaluated the applicability of our approach experimentally on three benchmarks: known-plaintext 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. The benchmarks contain instances with a substantial amount of xor-clauses to exhibit the potential benefits of xor-reasoning but also instances with few xor-clauses to demonstrate that the xor-reasoning module does not hinder the performance of the SAT solver even when it cannot reduce the number of heuristic decisions in the search. The results are promising as the number of heuristic decisions decreases with the help of the xor-reasoning module. Also, even though we believe there is still potential for improvement in data structures and algorithms of the proof-of-concept implementation, the solving times are comparable and on some larger instances the SAT solver minisat enhanced with xor-reasoning performs better than the unmodified version of minisat.

Our study on the research problem proposes further questions. The effect of the use of the xor-reasoning module on the decision heuristics of the SAT solver is an interesting subject. In principle, deduction performed by the SAT solver can be simulated in the xor-reasoning module when the constraints are encoded appropriately, so the number of heuristic decisions needed to solve an instance should be lower on average when the xor-reasoning module is used. This is the case typically, but on some instances our solver xor-minisat requires more heuristic decisions than the unmodified solver minisat. We believe that the decision heuristics employed by the SAT solver may cause this phenomenon.

We have introduced two methods for computing clausal explanations for conflicts in the xor-reasoning module that are likely to perform well. Comparing these to other methods like the method based on all-UIP-cuts presented in Section 3.7 is an interesting topic for future work.

An adaptive approach for performing xor-reasoning only when it is likely

to be beneficial according to a heuristic function could be a viable addition to the xor-propagation strategies proposed in this report. The presented xor-propagation strategies give an initial understanding of when xor-reasoning is beneficial to be performed and work fairly well on average. However, according to our experimental study, each xor-propagation strategy has a certain class of problems in which it performs best, that is, the optimal xor-reasoning strategy depends on the instance being solved.

Another interesting question is whether the search of the hybrid solver can be enhanced further by preprocessing the problem description, in particular, if more xor-clauses can be extracted from the problem description before the actual search. The solver march_eq does extensive preprocessing on the xor-clauses before starting the search. It is not evident whether preprocessing contributes to the effectiveness of the search method, but march_eq performs well with the Trivium benchmark so addressing how the xor-clauses could be preprocessed in order to enhance the overall performance of xor-minisat is a potential subject for future work.

Also, while performing promisingly well with regard to solving time, our proof-of-concept implementation still leaves room for algorithmic improvements which might make the approach presented in this report even more appealing.

Acknowledgements
I wish to thank Prof. Ilkka Niemelä and Docent Tommi Junttila for their insightful guidance. The report is based on my Master's thesis [20]. The financial support of the Academy of Finland (project 122399) is gratefully acknowledged.

REFERENCES

[1] *Data encryption standard.* U. S. Department of Commerce, National Bureau of Standards, 1977.

[2] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

[3] Gilles Audemard and Lakhdar Sais. Circuit based encoding of CNF formula. In Marques-Silva and Sakallah [24], pages 16–21.

[4] Fahiem Bacchus. Enhancing Davis Putnam with extended binary clause reasoning. In *Proceedings of the 18th AAAI Conference on Artificial Intelligence (AAAI-2002)*, pages 613–619. AAAI Press, 2002.

[5] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, chapter 26, pages 825–885. Volume 185 of Biere et al. [8], February 2009.

[6] Peter Baumgartner and Fabio Massacci. The taming of the (X)OR. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua

Sagiv, and Peter J. Stuckey, editors, *Computational Logic*, volume 1861 of *Lecture Notes in Computer Science*, pages 508–522. Springer, 2000.

[7] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):75–97, 2008.

[8] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[9] Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In Sokratis K. Katsikas, Javier Lopez, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006, Proceedings*, volume 4176 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2006.

[10] J. Chen. Building a hybrid SAT solver via conflict-driven, look-ahead and XOR reasoning techniques. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 298–311. Springer, 2009.

[11] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[12] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

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

[14] Marijn Heule, Mark Dufour, Joris van Zwieten, and Hans van Maaren. March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In Hoos and Mitchell [17], pages 345–359.

[15] Marijn Heule and Hans van Maaren. Aligning cnf- and equivalence-reasoning. In Hoos and Mitchell [17], pages 145–156.

[16] Marijn Heule and Hans van Maaren. Look-ahead based SAT solvers. In Biere et al. [8], pages 155–184.

[17] Holger H. Hoos and David G. Mitchell, editors. *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*. Springer, 2005.

[18] Paul Jackson and Daniel Sheridan. Clause form conversions for boolean circuits. In Hoos and Mitchell [17], pages 183–198.

[19] Matti Järvisalo and Tommi Junttila. Limitations of restricted branching in clause learning. In Christian Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *Lecture Notes in Computer Science*, pages 348–363. Springer, 2007.

[20] Tero Laitinen. Extending SAT solvers with parity constraints. Master's thesis, Aalto University School of Science and Technology, submitted, 2010.

[21] Tero Laitinen. XOR satisfiability solver module for DPLL integration. Student Project in Theoretical Computer Science, Aalto University School of Science and Technology, submitted, 2010.

[22] C. M. Li. Integrating equivalency reasoning into Davis-Putnam procedure. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 291–296. AAAI Press / The MIT Press, 2000.

[23] Panagiotis Manolios and Daron Vroon. Efficient circuit to CNF conversion. In Marques-Silva and Sakallah [24], pages 4–9.

[24] João Marques-Silva and Karem A. Sakallah, editors. *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *Lecture Notes in Computer Science*. Springer, 2007.

[25] João Marques-Silva. Practical applications of boolean satisfiability. In *Proceedings of the 9th International Workshop on Discrete Event Systems (WODES'08)*, pages 74–80. IEEE Press, 2008.

[26] Fabio Massacci and Laura Marraro. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 24(1/2):165–203, 2000.

[27] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[28] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[29] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Sais. Recovering and exploiting structural knowledge from CNF formulas. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, volume 2470 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2002.

[30] Christos M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[31] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *Journal on Software Tools for Technology Transfer*, 7(2):156–173, 2005.

[32] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *Proceedings of the 1996 International Conference on Computer-Aided Design, November 10-14, 1996, San Jose, CA, USA*, pages 220–227. ACM and IEEE Computer Society, 1996.

[33] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.

[34] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009.

[35] William Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson Education, 2002.

[36] Endre Süli and David Francis Mayers. *An Introduction to Numerical Analysis*. John Wiley & Sons, 1989.

[37] G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.

[38] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *Proceedings of the 2001 International Conference on Computer-Aided Design, November 4-8, 2001, San Jose, CA, USA*, pages 279–285. ACM, 2001.

[39] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In Andrei Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, volume 2392 of *Lecture Notes in Computer Science*, pages 295–313. Springer, 2002.

TKK REPORTS IN INFORMATION AND COMPUTER SCIENCE

TKK-ICS-R22 Antti E. J. Hyvärinen, Tommi Junttila, Ilkka Niemelä
Partitioning Search Spaces of a Randomized Search. November 2009.

TKK-ICS-R23 Matti Pöllä, Timo Honkela, Teuvo Kohonen
Bibliography of Self-Organizing Map (SOM) Papers: 2002–2005 Addendum.
December 2009.

TKK-ICS-R24 Timo Honkela, Nina Janasik, Krista Lagus, Tiina Lindh-Knuutila, Mika Pantzar, Juha Raitio
Modeling communities of experts. December 2009.

TKK-ICS-R25 Jani Lampinen, Sami Liedes, Kari Kähkönen, Janne Kauttio, Keijo Heljanko
Interface Specification Methods for Software Components. December 2009.

TKK-ICS-R26 Kari Kähkönen
Automated Test Generation for Software Components. December 2009.

TKK-ICS-R27 Antti Ajanki, Mark Billinghurst, Melih Kandemir, Samuel Kaski, Markus Koskela, Mikko Kurimo, Jorma Laaksonen, Kai Puolamäki, Timo Tossavainen
Ubiquitous Contextual Information Access with Proactive Retrieval and Augmentation.
December 2009.

TKK-ICS-R28 Juho Frits
Model Checking Embedded Control Software. March 2010.

TKK-ICS-R29 Miki Sirola, Jaakko Talonen, Jukka Parviainen, Golan Lampi
Decision Support with Data-Analysis Methods in a Nuclear Power Plant. March 2010.

TKK-ICS-R30 Teuvo Kohonen
Contextually Self-Organized Maps of Chinese Words. April 2010.

TKK-ICS-R31 Jefrey Lijffijt, Panagiotis Papapetrou, Niko Vuokko, Kai Puolamäki
The smallest set of constraints that explains the data: a randomization approach. May 2010.

ISBN 978-952-60-3223-8 (Print)
ISBN 978-952-60-3224-5 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)