JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von
Lee A. Barnett, MSc

Angefertigt am
Institut für Formale Modelle und Verifikation

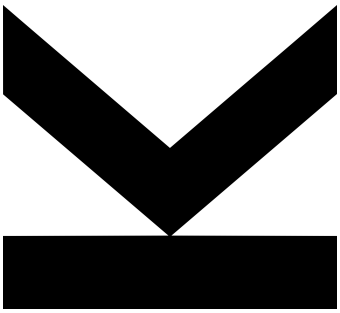
Betreuer und Erstbeurteiler
Univ.-Prof. Dr. Armin Biere

Zweitbeurteiler
Assoc. Prof. Marijn Heule

Mitbetreuung
Univ.-Prof.in Dr.in Laura Kovács

August 2022

Advanced Redundancy–Based Methods for SAT



Dissertation

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

im Doktoratsstudium

Technische Wissenschaften

JOHANNES KEPLER UNIVERSITÄT LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

The Boolean satisfiability (SAT) problem, deciding whether a formula in Boolean logic has a satisfying assignment, has become a fundamental problem in computer science and automated reasoning. Despite the absence of any known polynomial-time algorithm, SAT solving tools have become highly effective in practice, capable of handling large-scale instances from industrial applications. While the success of modern SAT solvers can be attributed to several factors, notable recent progress has been made by exploring the use of redundant clauses: disjunctions of literals that can be added to, or removed from, a formula while preserving satisfiability, though not necessarily logical equivalence. Current techniques utilizing redundant clauses, such as powerful proof systems and clause elimination procedures, show the potential of redundancy-based methods for SAT solving, but are not easily compatible with important solving procedures. This thesis explores limitations of current redundancy-based methods and presents novel approaches to redundancy-based reasoning in SAT.

We examine an important class of redundant clauses called covered clauses, which generalize multiple redundancy properties used by SAT solvers for clause elimination. Such formula simplification methods are demonstrably beneficial for solving large SAT instances from various application areas, though they may alter a formula's solution set, requiring a separate solution reconstruction process. Procedures based on covered clauses use a reconstruction process that is more complex and burdensome than similar procedures. We prove that for covered clauses, the partial assignments known as witnesses used by the typical solution reconstruction process can be as difficult to produce as solving the formula itself. We also show that covered clauses are not captured by the property used by the powerful propagation redundancy (PR) proof system. Furthermore, we develop a more flexible and general structure for witnesses to allow for efficient solution reconstruction for covered clauses as well.

This thesis also develops methods for reasoning about redundant but non-clausal Boolean constraints. Strong proof systems for certifying SAT solvers' results, and increasing trust in SAT technology, are based on the iterative addition of redundant clauses to a formula, but can struggle to express solving techniques that are not naturally in clause form. In particular, reasoning over "exclusive-or" (XOR) constraints embedded in a formula can afford tremendous gains in efficiency, but complicates the production of proofs, typically causing solvers to disable XOR reasoning when proofs are required. We generalize the notion of redundancy for clauses to redundancy for Boolean functions, retaining the strengths of clausal proof systems while expressing non-clausal reasoning. We use binary decision diagrams to represent redundant Boolean functions, producing proofs that capture XOR reasoning in a natural way. To show the effectiveness of this approach in practice, we present the results of a preliminary implementation of a proof checking tool which uses this method.

Kurzfassung

Das Erfüllbarkeitsproblem der Aussagenlogik (SAT) ist ein grundlegendes Problem der Informatik und des automatisierten Schließens. Die Kernaufgabe dabei ist zu bestimmen, ob es für eine gegebene aussagenlogische Formel eine erfüllende Belegung gibt. Trotz des Fehlens eines bekannten Polynomialzeit-Algorithmus haben sich Programme zur Lösung von SAT in der Praxis als sehr effektiv erwiesen und können große Formeln aus industriellen Anwendungen bewältigen. Während der Erfolg moderner SAT-Solver auf mehrere Faktoren zurückzuführen ist, wurden in letzter Zeit bemerkenswerte Fortschritte durch die Erforschung redundanter Klauseln erzielt. Solche Klauseln sind Disjunktionen von Literalen, die zu einer Formel hinzugefügt oder aus ihr entfernt werden können, ohne dass sich die Erfüllbarkeit, wenn auch nicht unbedingt die logische Äquivalenz, ändert. Aktuelle Techniken, die redundante Klauseln verwenden, wie z. B. starke Beweissysteme und Verfahren zur Vereinfachung von Formeln, zeigen das Potenzial von Redundanzmethoden für das SAT-Solving, sind aber nicht mit wichtigen Lösungsverfahren kompatibel. Diese Arbeit untersucht die Grenzen aktueller Redundanzmethoden und stellt neue Ansätze für redundanzbasiertes Schließen in SAT vor.

Wir untersuchen eine wichtige Klasse von redundanten Klauseln, die so genannten Covered Clauses, die mehrere Redundanzeigenschaften verallgemeinern, die von SAT-Solvern zur Klauseleliminierung verwendet werden. Solche Vereinfachungsmethoden sind nachweislich vorteilhaft für das Lösen großer SAT-Formeln aus verschiedenen Anwendungsbereichen, aber können die Modellmenge einer Formel verändern und erfordern deshalb einen Prozess zur Modellrekonstruktion. Verfahren für Covered Clauses verwenden einen Rekonstruktionsprozess, der komplexer und aufwändiger ist als ähnliche Verfahren. Wir beweisen, dass für Covered Clauses die Teilbelegungen, die als Zeugen bekannt sind und vom typischen Rekonstruktionsprozess verwendet werden, genauso schwierig zu erzeugen sein können wie die Lösung der Formel selbst. Wir zeigen auch, dass Covered Clauses nicht durch Propagationsredundanz (PR), eine der stärksten Redundanzeigenschaften, subsumiert wird. Darüber hinaus entwickeln wir eine flexiblere, allgemeine Struktur für Zeugen, um eine effiziente Modellrekonstruktion auch für Covered Clauses zu ermöglichen.

Weiterhin werden in dieser Arbeit Methoden für redundante, allgemeine aussagenlogische Constraints entwickelt. Starke Beweissysteme zur Zertifizierung der Ergebnisse von SAT-Solvern und zur Erhöhung des Vertrauens in die SAT-Technologie basieren aus dem iterativen Hinzufügen redundanter Klauseln zu einer Formel, haben aber Schwierigkeiten, Lösungstechniken auszudrücken, die nicht in Klauselform vorliegen. Insbesondere das Schließen über die in einer For-

mel eingebetteten „Exklusiv-Oder“ (XOR) Constraints kann enorme Effizienzgewinne ermöglichen, erschwert aber die Erstellung von Beweisen, so dass SAT-Solvern XOR-Techniken in der Regel deaktivieren, wenn Beweise erforderlich sind. Wir verallgemeinern Redundanz für Klauseln auf Redundanz für boolesche Funktionen, mit gleich starken Eigenschaften wie bei Klauselbeweissystemen. Wir verwenden binäre Entscheidungsdiagramme (BDDs), um redundante boolesche Funktionen darzustellen und Beweise für XOR-Techniken auf natürliche Weise zu erstellen. Um die Effektivität dieses Ansatzes in der Praxis zu zeigen, präsentieren wir die Ergebnisse einer vorläufigen Implementierung eines Proof-Checking-Tools, das diese Methode verwendet.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Acknowledgement

The completion of this thesis would not have been possible without a number of people. I would like to express my deep appreciation to my supervisor Armin Biere, whose advice has been indispensable and from whom I have learned an immeasurable amount. I am also grateful to Marijn Heule for reviewing this thesis, and to the other members of my committee for their careful attention to its content.

I would like to extend my sincere gratitude to the Linz Institute of Technology AI Lab and the State of Upper Austria, as well as the LogiCS Doctoral College and the Austrian Science Fund, for their generous support throughout my studies. I am also grateful to my colleagues and coworkers in Linz, especially Doris Falb, whose help and kindness greatly improved my time in Austria.

This thesis is certainly thanks, as well, to David Plaisted, who introduced me to automated reasoning, and who taught me how to learn and how to teach.

I am extremely thankful for the support and encouragement of my friends and family. I am also thankful to our wonderful dogs, Cheddar and Brie, for being exceptional officemates throughout the pandemic. Finally, to Courtney: thank you for bearing with me, and for everything you do.

Contents

List of Figures

Chapter 1

Introduction

The Boolean satisfiability problem, or SAT, has become a fundamental representation in computer science and a key area in the field of automated reasoning. In short, it asks for the possible truth values of formulas in classical propositional logic: expressions composed of Boolean variables, which can be only *true* or *false*, and simple operations such as *and*, *or*, and *not*. SAT is well known as the prototypical NP-complete problem, which means many computational questions about important properties of graphs, strings, and puzzles can be interpreted and solved as an instance of SAT, but also that no polynomial-time algorithm for solving SAT is known to exist.

In practice, however, computer programs for solving SAT instances are often immensely efficient and capable of handling formulas involving even millions of variables coming from industrial application domains, such as circuit verification [24, 40, 61], automated planning [62, 82], and logical cryptanalysis [73, 74, 87]. Additionally, SAT solvers are often components of other reasoning engines, such as SMT, or satisfiability modulo theories, solvers [9] and automated theorem provers for first-order logic (for example, [96]). These tools each have their own application areas, so in this sense the reach of SAT extends even further; for example, SMT solvers have been used in the area of cloud security [5].

The effectiveness of SAT solvers at scale is the result of decades of research and engineering effort which have led to clever search algorithms such as conflict-driven clause learning, or CDCL [10, 72], formula simplification procedures such as variable elimination [31], and careful solver implementation techniques and data structures, such as watched literals [75]. Recently, significant progress in the design of SAT solving techniques has been made possible by the idea of reasoning about redundancy, in contrast to reasoning only about entailment. A formula A entails a formula B if B evaluates to *true* under any assignment of values to variables for which A is *true*, in which case the combined formula A *and* B is logically equivalent to A alone. A program tasked with solving A may then choose to instead solve A *and* B, especially if the inclusion of B can expedite solving by making other entailments easier to find. On the contrary, if the program is to solve A *and* B, it could opt to use the simpler and shorter formula A.

Yet the specific task of a SAT solver is to determine whether a formula is satisfiable, which means it is *true* under some assignment. As such it is not strictly necessary that A *and* B be equivalent to A in order to replace one by the other, but only that conjoining B to A would not change the answer to the SAT instance; that is, whether it is satisfiable. In this case B is said to be redundant with respect to A [64]. Redundancy-based reasoning, which has also been called interference-based reasoning [43], is focused on formula alterations, such as adding or removing clauses, that are satisfiability-preserving.

Redundancy has been impactful in SAT in two complementary ways. First, the removal of redundant clauses from formulas, which are typically presented in conjunctive normal form as a list of clauses, is the basis of clause elimination methods, such as blocked clause elimination [59, 67]. These are formula simplification techniques that identify and remove clauses meeting some efficiently-decidable condition, called a redundancy property, that ensures the clauses can be removed without changing the formula's satisfiability. Clause elimination is typically used during preprocessing, before performing core algorithms such as CDCL, or at various points throughout solving, sometimes referred to as inprocessing [60]. Methods that remove redundant clauses, especially used in combination with other simplification methods, can be beneficial to solving, enabling large formulas to be solved more efficiently [47, 54].

Second, redundancy properties form the basis of proof systems used by SAT solvers to enable certification of their results. SAT solvers are employed to solve problems from critical application areas so it is crucial that their output can be trusted. If a formula is found to be satisfiable, it is usually easy for the solver to provide an assignment, such as by listing each variable and the assigned truth value, on which the formula can be evaluated, independently and efficiently, to verify that the result is indeed *true*. On the other hand, if a formula is found to be unsatisfiable, the solver must produce a proof that clearly demonstrates the formula always evaluates to *false*. This is primarily done by providing a sequence of clause addition and deletion instructions, ending with the addition of the unsatisfiable empty clause, where each clause to be added meets the conditions of some easily checked redundancy property with respect to the accumulated formula; that is, the result of applying all prior instructions to the original formula. The popular DRAT proof system [99], for example, has become standard in SAT solving, and has been the proof format used by the SAT competition in recent years [6]. It is based on the RAT property [60], which can express the reasoning behind many of the techniques commonly used by SAT solvers, including those not easily expressed in the traditional propositional resolution system. In fact, proof systems based on strong redundancy properties, such as Propagation Redundancy (PR) [51], admit exponentially more compact proofs for some formulas than are possible with resolution (see [20]).

The initial success of clause redundancy methods, both for clause elimination and strong proof systems, shows the significant potential of redundancy-based reasoning in SAT solving (see also [43]). However, current approaches to redundancy are limited and can be difficult to generalize, restricting their usefulness and complicating the realization of this potential. This thesis highlights ways in which such limitations occur and presents a more general and flexible formulation of redundancy, and redundancy properties, to overcome these limitations, with the aim of advancing the effectiveness of redundancy-based reasoning in SAT solving. The remainder of this introductory chapter provides a broad overview of the main topics and contributions of this thesis, after briefly covering some background.

1.1 Background

The chapters in this thesis are mostly self-contained, providing the preliminaries necessary to understand the included content. This section offers general background information that is intended to be helpful in understanding the contributions of this thesis.

1.1.1 Propositional Satisfiability

Propositional logic has long been established as a foundational element of reasoning in mathematics and philosophy, dealing with the ways declarative sentences can be joined or altered to form more complex statements, and the properties of these ways. The statement "Linz is the capital of Upper Austria, and lies on the Danube" can be understood as containing two simpler propositions: that A, "Linz is the capital of Upper Austria," and B, "Linz lies on the Danube," joined as A *and* B. This symbolic abstraction of the English-language statement has the advantage that it also represents many other statements, such as "Today is Tuesday, and the program P compiles without error" or "$4 > 3$ and $6 > 7$." The language of propositional logic allows any combinations of propositions by sentential operators, such as conjunction ("and," written $\wedge$), disjunction ("or," written $\vee$), and negation ("not," written $\neg$).

The truth of such a combination is defined by the truths of the propositions: $A \wedge B$ is true only in case A and B both are true statements, and it is false if either A or B is false. Yet for certain combinations, due to their form their truth does not depend at all on the underlying propositions; for instance, $(\neg A \vee B) \vee A$ is true, and $B \wedge \neg B$ is not, regardless of what is meant by A and B. The SAT problem can be seen as asking about a version of this distinction: given a statement S in propositional logic, however complex, is there a way to assign meaning to the propositions in S so that S is true, or is S always false, as a consequence of its form?

More formally, we represent atomic propositions by Boolean variables, which have two possible values: *true* and *false*, often represented by 1 and 0, respectively. An assignment to a set of Boolean variables fixes each variable in the set to a particular value. Boolean expressions, or formulas, are constructed by applying logical operations, such as negation, conjunction, and disjunction, to variables or to other expressions. Well-formed formulas evaluate to either *true* or *false* for each assignment to its variables, defining a function from the set of all possible assignments to their corresponding truth values. A formula is satisfiable if it evaluates to *true* for some assignment, otherwise it is unsatisfiable and evaluates to *false* for every assignment. The SAT problem is to decide, given a Boolean formula, whether it is satisfiable.

The NP-completeness of this question was proven independently by Cook [25] and Levin [69], showing that any problem solvable by a non-deterministic algorithm in polynomial time can be solved as an instance of SAT. One consequence of this is that it is widely conceded there is likely no deterministic, polynomial-time algorithm for SAT. Despite this, modern SAT solving programs often decide instances of SAT extremely efficiently, leading to the increasing use of SAT solvers as general-purpose problem solving tools for the many interesting and useful computational problems belonging to NP. Even so, there are many problems on which SAT solvers perform poorly (see also [35]), and the questions about why some formulas present more issues than others, and how these issues can be overcome, are central in SAT solving research.

1.1.2 Clause Elimination

SAT solvers typically expect the formulas they take as input to be in conjunctive normal form (CNF): a conjunction of one or more clauses, which are disjunctions of Boolean literals[1]. Many challenging SAT formulas contain a large number of clauses, whether as a consequence of being expressed in CNF or due to the nature of the particular instance. Formula simplification procedures are crucial to efficiently solving large formulas, especially those derived from industrial application areas [47, 54]. There is a large body of work on the efficacy of these methods during preprocessing (see [15]). Further, formula simplification, especially combinations of multiple methods, can be incredibly powerful as inprocessing techniques, applied at various stages throughout solving rather than just at the beginning [60].

Clause elimination procedures form a particular class of formula simplification methods, and are designed to reduce the size of formulas in CNF by removing clauses that can be deemed unnecessary. If a formula F entails a clause C, written $F \vDash C$, then the inclusion of C in the formula $F \wedge C$ can be considered unnecessary

[1] We often represent a clause by the set of its literals, for example $\{a, \neg b\}$ for $a \vee \neg b$, and a formula by the set of its clauses.

in the sense that any solutions found for F will also satisfy C, and thus $F \wedge C$, anyway. Deciding whether $F \models C$ in general is co-NP-complete, but there are efficiently-decidable conditions that ensure a clause is entailed. For example, if C includes both a literal l and its negation $\neg l$, then C is a tautology and will be satisfied by any assignment; that is, C is entailed by any F. If F includes a clause C' such that $C' \subseteq C$, then C is called a subsumed clause in F and $F \models C$, as C is satisfied by any assignment satisfying C'. Tautology elimination and subsumption elimination are common and useful procedures in SAT solving that can reduce the size of formulas by identifying and removing tautological and subsumed clauses [31]. Both can be called equivalence-preserving, or model-preserving, as F and F' are logically equivalent, written $F \equiv F'$, for the original formula F and reduced formula F'.

Not all clause elimination procedures are equivalence-preserving, and may remove clauses from a formula F which are not guaranteed to be satisfied by assignments satisfying the rest of F. A clause C is blocked in F if there is a literal $l \in C$ meeting the following condition: for every clause $\neg l \vee D \in F$, there is another literal $k \in C$ such that $\neg k \in D$ [67]. For example, in the formula

$$F = (\neg a \vee \neg c) \wedge (a \vee \neg b \vee c) \wedge (\neg a \vee b \vee \neg d)$$

the middle clause $C = (a \vee \neg b \vee c)$ is blocked by the literal a. Blocked clause elimination [59] can remove C, leaving only two clauses in the reduced formula $F' = (\neg a \vee \neg c) \wedge (\neg a \vee b \vee \neg d)$. However, $F' \not\equiv F$. To see this, consider the assignment τ that sets b to *true* and all other variables to *false*; τ satisfies both clauses in F', as they each contain the literal $\neg a$ which evaluates to *true*, but τ falsifies all literals in the removed clause C. This means τ satisfies F' but not F, and that the removal of C was not an equivalence-preserving simplification.

Procedures that remove, or add, blocked clauses can change the set of solutions to a formula, but they cannot change whether solutions exist. For the example above, even though τ does not satisfy F, the assignment τ' that assigns both a and b to *true*, and falsifies the other variables, does satisfy F, meaning F' and F are both satisfiable formulas. In other words, the original formula F and the reduced formula F' are satisfiability-equivalent. Removing blocked clauses will not cause an unsatisfiable formula to become satisfiable, as blocked clauses are redundant. If a clause C can be added to, or removed from, a formula F without affecting its satisfiability, then C is redundant with respect to F. Clause elimination procedures are typically based on redundancy properties, such as the existence of a blocking literal, that can be quickly checked to ensure a clause is redundant (see [54]).

Removing redundant clauses from a formula is safe in that it will not change the formula's satisfiability, but, as seen, it can change the formula's specific set of satisfying assignments. Solution reconstruction is an important process

that transforms assignments satisfying the reduced formula into assignments that satisfy the original formula, which are often required by users of SAT solvers, for example as evidence of the formula's satisfiability. This can be achieved by recording a partial assignment called a witness for each redundant clause removed by some elimination procedure. Intuitively, a witness for a clause C explains how assignments falsifying C can be adjusted to satisfy it, without accidentally falsifying other clauses in the formula.

More precisely, a partial assignment is a set ω of non-contradictory literals, so that $l \in \omega$ means $\neg l \notin \omega$, and can be applied to reduce clauses and formulas. Specifically $C|_\omega = \top$, a satisfied clause, if ω includes a literal in C, otherwise $C|_\omega = \{l \in C \mid \neg l \notin \omega\}$, and likewise if $C|_\omega = \bot$, the unsatisfiable empty clause, for some $C \in F$ then $F|_\omega = \bot$, otherwise $F|_\omega = \{C|_\omega \mid C \in F \text{ and } C|_\omega \neq \top\}$. A partial assignment is a witness for C with respect to F if $C|_\omega = \top$, and further $F'|_\alpha \vDash F'|_\omega$, where $\alpha = \{\neg l \mid l \in C\}$, where F' is the formula F without C.

In the example above we removed the clause $C = (a \vee \neg b \vee c)$ as it is blocked by the literal a. The partial assignment $\omega = \{a, b, \neg c\}$ is a witness for C, as $C|_\omega = \top$, and further $F'|_\omega = \top$ is entailed by $F'|_\alpha$, for the reduced formula F' without the clause C. Then the assignment $\tau = \{a \to \mathit{false}, b \to \mathit{true}, c \to \mathit{false}, d \to \mathit{false}\}$ which falsifies C can be altered by taking the values indicated by ω: fixing a and b to *true*, and c to *false*, leaves $\tau' = \{a \to \mathit{true}, b \to \mathit{true}, c \to \mathit{false}, d \to \mathit{false}\}$. As seen already, τ' satisfies C and thus F.

This process of altering assignments using witnesses can be extended to a list of redundant, removed clauses using the reconstruction function. Given a witness for each clause, the reconstruction function checks, in turn, each removed clause C to see if it is satisfied by the current assignment, adjusting it using the witness if not [32, 60]. As a witness exists for any redundant clause [51], and most clause elimination procedures identify witnesses for the clauses they remove, this provides a uniform way to reconstruct solutions to the original formula after removing any number of redundant clauses.

1.1.3 Proofs of Unsatisfiability

While efficiently checkable redundancy properties are at the heart of clause elimination procedures, they are also a foundation for strong proof systems used by SAT solvers, enabling the efficient certification of SAT results. It is important for users of SAT solvers to know that their results are correct without needing to trust blindly that the solver made no mistakes. When a formula is found to be satisfiable, it is typically straightforward for the solver to report a satisfying assignment as evidence of this result; even after using clause elimination procedures that are not equivalence-preserving, the reconstruction function can be used as described above. If instead a solver reports that a formula F has

no solutions, it should provide a proof of the unsatisfiability of F that can be independently and efficiently validated.

A traditional choice of how a solver can prove that F is unsatisfiable is to provide a refutation of F in the propositional resolution proof system, also called a resolution proof of the unsatisfiability of F. Proofs in this system are based on the resolution rule which, from two clauses $l \vee C$ and $\neg l \vee D$ with complementary literals l and $\neg l$, derives their resolvent $C \vee D$ [83]. This may also be called the l-resolvent of $l \vee C$ and $\neg l \vee D$, and written $l \vee C \otimes_l \neg l \vee D$. Multiple formats have been described for using resolution proofs in practice (for example, [14, 100]), but generally a proof can be seen as a list of clauses $C_1, \ldots, C_n$, such that C_n is the empty clause $\bot$, and each C_i is the resolvent of two clauses in the set $\{F, C_1, \ldots, C_{i-1}\}$, possibly annotated to indicate for each C_i exactly which two clauses can be resolved to produce it [37].

While resolution is a convenient system for certain purposes other than validating unsatisfiability results, such as generating interpolants [95] and extracting minimal unsatisfiable cores [30], the production of resolution proofs can be cumbersome. The CDCL algorithm, used by many solvers, extends a formula by learning new clauses while searching for solutions; each learned clause can be derived by a sequence of resolution steps. These steps must be explicitly recorded to provide a resolution proof, and further it can be difficult to provide such steps in combination with the use of learned clause minimization strategies [90]. This is made easier by enabling proof checking tools to perform unit propagation, which is the repeated replacement of F by $F|_l$, for each unit clause $(l) \in F$. Then checking individual resolution steps can be replaced by checking that each clause meets the conditions of RUP, or Reverse Unit Propagation, where a clause C is RUP with respect to a formula F if unit propagation on $F \wedge \neg C$ produces the empty clause [39]; this is also written $F \vdash_1 C$. Any clause learned by a solver during CDCL is a RUP clause and can be added to a RUP proof: a list of clauses $C_1, \ldots, C_n$, such that C_n is the empty clause $\bot$, and each C_i is RUP with respect to $F \wedge C_1 \wedge \cdots \wedge C_{i-1}$.

Resolution and RUP both only derive clauses entailed by the formula, so they are not easily compatible with techniques that use only satisfiability-preserving reasoning. The RAT redundancy property allows the derivation of clauses not entailed by the formula, where a clause $l \vee C$ is RAT, or is a Resolution Asymmetric Tautology [60], with respect to F if $F \vdash_1 C \vee D$ for each clause $\neg l \vee D \in F$. For example, with respect to the formula

$$F = (\neg a \vee \neg d) \wedge (a \vee \neg b \vee c) \wedge (\neg c \vee \neg d)$$

the clause $C = (a \vee \neg b)$ is RAT for the literal $l = a$: only the first clause in F includes the literal $\neg a$, and the clause $(\neg b \vee \neg d)$ is indeed a RUP clause in F. Many redundancy properties used in SAT solving are directly expressed by

RAT, leading to the emergence of the DRAT system, which also permits proof steps deleting clauses, as the modern standard for proofs of unsatisfiability; for example, output in the DRAT format is required for certain tracks of the SAT competition in recent years [6].

More recently, a generalization of RAT known as PR, for Propagation Redundancy, was shown to admit surprisingly concise proofs for certain families of formulas considered hard for the resolution proof systems. This property is based on a restriction of the witness characterization of redundant clauses; recall that a clause C is redundant with respect to a formula F if it has a partial assignment, or witness, ω such that $C|_\omega = \top$ and $F|_\alpha \vDash F|_\omega$. While this entailment requirement is likely not efficiently decidable, a clause C is PR if it has a witness ω such that each clause in $F|_\omega$ is RUP with respect to $F|_\alpha$, also written $F|_\alpha \vdash_1 F|_\omega$.

One example of a collection of formulas having short PR proofs are the pigeonhole formulas. The formula PHP_n^m expresses the statement that there is a 1-to-1 mapping of m pigeons into n holes, and is thus unsatisfiable for $m > n$. More precisely, it includes the variables $x_{i,j}$, where an assignment in which $x_{i,j}$ is fixed to *true* can be thought as a mapping in which pigeon number i is in hole number j. Then the formula comprises the clauses:

$$x_{i,1} \vee \cdots \vee x_{i,n}$$

for every $i \leq m$, and the binary clauses $\neg x_{i,k} \vee \neg x_{j,k}$ for all $i \neq j$ up to m and $k \leq n$. Any resolution proof of PHP_{n-1}^n requires length $2^{\Omega(n)}$ [42], each step of which must be output by a solver producing resolution proofs. On the other hand, there exist polynomial-length PR proofs for PHP_{n-1}^n [51], and for several related formulas (see [20] for more examples).

It is important to note that if resolution is augmented to allow proof steps that define new variables, resulting in the Extended Resolution proof system, then polynomial-length proofs of PHP_{n-1}^n are possible [42, 92]. In fact, Extended Resolution is a very strong proof system, and there are at present no known lower-bounds on the lengths of extended resolution proofs. Though some SAT solving methods have been designed to produce proofs in extended resolution [4, 71], it is in general not clear how to define new variables as to take advantage of the strengths of this system. On the other hand, there has been some initial success with learning PR clauses while solving to produce short PR proofs [50].

1.2 Overview

In this section we provide an overview of the main contributions of this thesis. These contributions are separated by the chapter in which they are presented.

This thesis includes both new material and work that appears in peer-reviewed proceedings of major international conferences.

- Chapter 2 is included in the proceedings of the 10th International Joint Conference on Automated Reasoning (IJCAR) [8], the flagship, biennial conference on all aspects of automated reasoning.
- Chapter 3 is included in the proceedings of the 28th International Conference on Automated Deduction (CADE) [7], a renowned, A-ranked conference in this field.
- Chapter 4 has not been previously published, and is new to this thesis.

Specific publication information is listed at the beginning of each chapter.

Chapters 2 and 3 were written by the present author in collaboration with co-authors. For these chapters, this section provides a statement in order to indicate the specific contributions for which the author of this thesis is responsible.

Chapter 2: Covered Clauses Are Not Propagation Redundant

This chapter revisits a procedure called covered clause elimination, and investigates the limits of PR. As the title suggests, it demonstrates that the complex redundancy property underlying covered clauses is not captured by the highly general PR property. The chapter also provides a concrete algorithm for deciding whether clauses are covered, and for reconstructing solutions to satisfiable formulas after covered clauses have been eliminated.

Covered clause elimination (CCE) [48] is a formula simplification procedure that tries to identify and remove covered clauses, a class of redundant clauses based on a complex property that generalizes other well known properties, including the blocked property.

Recall that a literal k *blocks* a clause $k \vee C$ in a formula F if, for every clause $\neg k \vee D$ in F, there is some literal in D whose negation occurs in C; that is, if every k-resolvent between $k \vee C$ and a clause in F is tautological [67]. If some of these resolvents are not tautological then k does not block the clause. However, if any literals occur in every one of these non-tautological resolvents, they are *covered* by k. For example, consider the clause $k \vee a \vee b$ and

$$F = (\neg k \vee \neg b) \wedge (k \vee c) \wedge (\neg k \vee \neg c \vee l) \wedge (\neg k \vee l) \wedge (c \vee d).$$

The resolvent of $k \vee a \vee b$ with $\neg k \vee \neg b$ is tautological, but its resolvents with $\neg k \vee \neg c \vee l$ and $\neg k \vee l$ both include the literal l, which does not already occur in $k \vee a \vee b$. As a result, k covers l.

The existence of covered literals means that the clause C can be extended by adding these literals without affecting the formula's satisfiability. This is because if some k in C covers a literal l, then C is redundant with respect to $F \wedge (C \vee l)$, so that the formulas $F \wedge C$ and $F \wedge (C \vee l)$ are equisatisfiable. Covered clause elimination begins by, for a clause C, iteratively adding to it any literals l covered

by some $k \in C$. If this extension of the original clause C is blocked, then C is a *covered clause* and can be removed by CCE, as it is redundant with respect to F [48].

More precisely, this chapter considers asymmetric covered clause elimination (ACCE), which also allows the addition of literals l to C that are *asymmetric* to C in F; that is, there is some clause $C' \vee \neg l$ in F, where $C' \subseteq C$. The addition of asymmetric literals to a clause is in fact equivalence-preserving, so that asymmetric literals can be added to a clause C in F without affecting its set of satisfying assignments [47]. Asymmetric literals added to C do not themselves cover any new literals, but their presence can allow new covered literals to be added by other elements of C. As a result, ACCE is a stronger clause elimination procedure than CCE.

CCE and ACCE were described mathematically by Heule, Järvisalo, and Biere [48], but without an explicit algorithm for identifying such clauses. Chapter 2 provides, and proves correct, an algorithm for identifying asymmetric covered clauses. This algorithm also produces, for any asymmetric covered clause, a sequence of witness-labeled clauses which can be used in the typical reconstruction function to reconstruct solutions to it.

This sequence used for solution reconstruction is notably different than how solution reconstruction is performed for other redundant clauses, such as for clauses removed by blocked clause elimination. Instead of recording a witness for each clause C removed by ACCE, so that assignments not satisfying C can be corrected in a single step by the reconstruction function, a number of intermediary witnesses and clauses are recorded. For instance, the clause $C = (a \vee b)$ is covered with respect to the formula

$$F = (\neg a \vee \neg b \vee c) \wedge (\neg a \vee x) \wedge (\neg c \vee \neg x \vee y) \wedge (\neg x \vee y) \wedge (\neg b \vee \neg y).$$

This is because the literal a in C covers x and the literal x in $(a \vee b \vee x)$ covers y, so that the extended clause is $C_{\text{EXT}} = (a \vee b \vee x \vee y)$. Then C_{EXT} is blocked by y, producing the sequence of of witness-labeled clauses:

$$(\{\neg a, \neg b, \neg x, y\} : a \vee b \vee x \vee y)$$
$$(\{\neg a, \neg b, x\} : a \vee b \vee x)$$
$$(\{a, \neg b\} : a \vee b).$$

Given this sequence, the reconstruction function corrects assignments to ensure each extension of C is satisfied, ending with C itself. We provide an example to show that the size of this sequence can be quadratic in the length of the extended clause. For other clause elimination procedures, reconstruction can be done with a single witness is at most linear in the size of the clause; for example,

to reconstruct solutions for clauses removed by blocked clause elimination, only the blocking literal must be recorded for each clause.

Although ACCE uses a complex sequence of witnesses to reconstruct solutions for each clause C it removes, rather than a witness for C itself, it has been proven that there exists a single witness for every redundant clause [51]. This raises the question of how to formulate witnesses for covered clauses. However, we prove in Proposition 2.4.1 that finding a single witness for a covered clause can be as difficult as finding solutions to formulas in general. In short, this is because the witness may need to include a satisfying assignment for large parts of the formula F. Moreover, many covered clauses are not PR: the proof of Theorem 2.4.1 provides the example of the clause $C = (k \vee l)$ and the formula

$$\begin{aligned} F \;=\; & (x \vee \neg k) \;\wedge\; (\neg x \;\vee\; \neg y) \;\wedge (y \vee \neg l) \;\wedge \\ & (x \vee a \vee b) \wedge (x \vee a \vee \neg b) \;\wedge\; (x \vee \neg a \vee b) \wedge (x \vee \neg a \vee \neg b) \;\wedge \\ & (y \vee c \vee d) \wedge (y \vee c \vee \neg d) \;\wedge\; (y \vee \neg c \vee d) \wedge (y \vee \neg c \vee \neg d). \end{aligned}$$

The literal k in C covers x, and the literal l covers y, leading to the extended clause $C_{\text{EXT}} = k \vee l \vee x \vee y$. This clause is blocked in F, but there is not a witness for C that satisfies the PR property.

PR encompasses many redundancy properties, so it is surprising that some covered clauses are not PR. Of course, PR is less general than redundancy itself, so it is not unexpected that there exist redundant clauses not meeting the conditions of PR. Yet the particular difficulty raised by covered clauses is interesting as it highlights the limits of not just PR, but the partial assignment structure used for witnesses in the characterization of clause redundancy in general.

This observation leads Chapter 2 to also consider clause redundancy itself, rather than individual redundancy properties such as PR or covered clauses. Specifically, Theorem 2.5.1 proves that deciding whether a clause C is redundant with respect to a formula F is a complete problem for the complexity class co-D^{P}. The class D^{P} was defined by Papadimitriou and Yannakakis to classify problems that are hard for both NP and co-NP, but do not seem to be complete for either, and is the class of problems that are the intersection of a problem in NP and a problem in co-NP [79]:

$$\mathrm{D}^{\mathrm{P}} = \{L_1 \cap L_2 \mid L_1 \in \text{NP and } L_2 \in \text{co-NP}\}.$$

This chapter is focused on covered clause elimination, a strong clause elimination procedure, but also investigates the limits of how redundancy is characterized, especially the use of partial assignments as witnesses, as well as the PR property. Its results suggest that the exploration of novel, and more flexible, definitions of redundancy could be impactful in SAT.

Author Contributions

The proof of correctness of the presented ACC algorithm was written by the present author. The formulation of the witness problem for a redundancy property, the proof that finding witnesses for covered clauses solves the search analog of the SAT problem, and the proof that covered clauses are not all propagation redundant, are also the work of the present author. The formulation of Theorem 2.5.1 was the result of several discussions among the co-authors, while the proof was written by the present author.

Chapter 3: Non-Clausal Redundancy Properties

In Chapter 3 we extend the use of redundant clauses in SAT by exploring redundancy properties for constraints that are not in clause form, developing a more general notion of redundancy. It is motivated both by the issues detailed in Chapter 2, and the limits of clause redundancy-based proof systems.

The system in which a SAT solver produces proofs places restrictions on the techniques the solver may use, as those proofs must be able to express the reasoning steps taken by the solver. As modern SAT solvers primarily use proof systems such as DRAT based on clause redundancy properties, they are well suited for capturing reasoning performed by SAT procedures that operate at the clausal level. While many SAT procedures are naturally clausal, there are powerful and efficient techniques that are difficult to express this way. An important example is reasoning about "exclusive-OR," or XOR, constraints. SAT instances from application domains such as logical cryptanalysis and arithmetic circuit verification often encode many XOR constraints [89], each of which can require a large set of clauses to express in CNF. For example, the straightforward translation into CNF of the expression a XOR b XOR c, which is *true* only if an odd number of the variables a, b, c are assigned the value *true*, is

$$(\neg a \vee \neg b \vee c) \wedge (\neg a \vee b \vee \neg c) \wedge (a \vee \neg b \vee \neg c) \wedge (a \vee b \vee c).$$

For formulas including clauses for more than a few XOR constraints, it is typically beneficial for solvers to use specialized procedures that extract them from the formula, by detecting sets of clauses that encode XORs, and perform reasoning on them directly, for example during preprocessing (see [15, 88, 89, 97]). A typical approach is to interpret the extracted XORs as a system of linear equations, which can be simplified using matrix techniques such as Gaussian elimination [89]. This kind of reasoning is difficult to capture with clause redundancy properties such as RAT and PR; while translations have been described, for example by Philipp and Rebola-Pardo [80], they are not typically implemented due to their complexity. Instead, solvers choose to disable XOR reasoning when

proof production is required, in some cases dramatically worsening their performance.

This can be illustrated by considering the Tseitin formula [92, 94] for certain families of graphs. The Tseitin formula for a graph G is constructed by first assigning an odd number of vertices in G the label 1, sometimes called a "charge," and the remaining vertices the label 0; the formula is then a set of clauses expressing that, for each vertex, the parity of its incident edges match its label. By a classic result of Euler, the sum of the degrees in a graph is even, so that this formula is unsatisfiable. However, for particular sets of graphs, these formulas have no polynomial-size resolution proofs [92, 94], yet they encode XOR constraints and can be proven unsatisfiable by a short sequence of Gaussian elimination reasoning steps. As CDCL essentially searches for proofs in resolution [11], it is crucial that solvers can perform other procedures that surpass the abilities of CDCL alone.

While systems based on strong clause redundancy properties, such as PR, also include polynomial-size proofs for many hard formulas, including Tseitin formulas, it is not clear how to design general solving methods that take full advantage of these strengths (although there has been some initial success in this direction; see [50]). At the same time, Gaussian elimination reasoning and other non-clausal reasoning techniques are established and efficient solving strategies. In this work we aim to extend the proof systems used by modern SAT solvers to be compatible with non-clausal reasoning, in order to enable such powerful solving methods even when proof output is required.

In order to generalize redundancy-based methods beyond clauses, we present a characterization of redundancy for Boolean functions, where a function g is redundant with respect to a function f if the functions f and $f \wedge g$ are satisfiability-equivalent: both satisfiable, or both unsatisfiable. This is aided by generalizing the notion of witness as used in redundancy: instead of applying a partial assignment to a formula or clause, we define transformations as functions which map assignments to assignments, so that a transformation σ can be composed with a function f to create a new Boolean function $f \circ \sigma$. Then we prove as Theorem 3.3.1 that g is redundant with respect to f if and only if there exists a transformation ω which acts as a witness in precisely the way as partial assignments can act as witnesses. More precisely, this means $g \circ \omega$ is the tautological function mapping every assignment to *true*, and

$$f \circ \alpha \vDash f \circ \omega$$

where α is a transformation ensuring that $g(\alpha(\tau))$ is *false* for any assignment τ.

We use this characterization as a framework to define redundancy properties for Binary Decision Diagrams (BDDs), graphs expressing Boolean functions [3, 17, 68], for which XOR constraints in particular have compact representations.

We use a BDD operation known as *constrain* to compute a generalized cofactor $f|_g$ of a BDD f by another g, and define an operation UnitProp for performing unit propagation over a collection of BDDs. This allows BDD redundancy properties $\mathsf{RUP}_{\mathrm{BDD}}$ and $\mathsf{PR}_{\mathrm{BDD}}$ to be defined analogously to RUP and PR. For example, a BDD g is $\mathsf{RUP}_{\mathrm{BDD}}$ with respect to the conjunction of BDDs $f_1 \wedge \cdots \wedge f_n$ if $\mathsf{UnitProp}(f_1|_{\neg g}, \ldots, f_n|_{\neg g})$ produces the always *false* BDD, or produces two BDDs which are the negations of one another.

Finally, we demonstrate that proof systems based on these BDD redundancy properties have the same strengths of their corresponding clausal systems, but also easily express Gaussian elimination reasoning. The steps performed by Gaussian elimination can be captured by an inference rule which derives the sum of XOR constraints; for example, if X is the expression a XOR b and Y is the expression b XOR c then their sum $X \oplus Y$ is $\neg a$ XOR c. We show that the BDD for $X \oplus Y$ is $\mathsf{RUP}_{\mathrm{BDD}}$ with respect to the two BDDs for X and Y, and in Theorem 3.5.1 prove that any sequence of Gaussian elimination reasoning steps can be translated into a sequence of $\mathsf{RUP}_{\mathrm{BDD}}$ addition steps without significantly increasing its size.

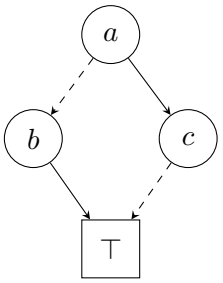
Chapter 3 also presents the results of our tool dxddcheck, a preliminary implementation in Python of a proof checker using the $\mathsf{RUP}_{\mathrm{BDD}}$ redundancy property. In brief, we use the SAT solver Lingeling [13] on formulas that can be found to be unsatisfiable using Gaussian elimination reasoning. From its output we construct sequences of XORs summarizing the Gaussian elimination steps, from which dxddcheck constructs the corresponding BDDs and checks that each new XOR meets the conditions of the $\mathsf{RUP}_{\mathrm{BDD}}$ property. Even without specialized data structures or other optimizations, the proofs can be checked relatively quickly, showing the promise of using BDDs, and non-clausal reasoning more generally, for proof production in SAT solvers.

Author Contributions

All lemmas, propositions, and theorems included and proven in this chapter, as well as the examples, are the work of the present author. The UnitProp algorithm and the properties $\mathsf{RUP}_{\mathrm{BDD}}$, $\mathsf{RUP}_{\mathrm{path}}$, and $\mathsf{PR}_{\mathrm{BDD}}$, were formulated and written by the present author, as was the proof checking tool dxddcheck. Experiments were carried out in collaboration with the co-author.

Chapter 4: Redundancy by Transformation

Chapter 4 expands on the concept of transformations, functions that map assignments to assignments as introduced in Chapter 3, showing how transformations can be used as witnesses for redundant clauses. Importantly, we present compactly expressible transformations that can act as witnesses for covered clauses,



$\tau(a)$	$\tau(b)$	$\tau(c)$	$\gamma(\tau)(a)$	$\gamma(\tau)(b)$	$\gamma(\tau)(c)$
0	0	0	0	1	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	0
1	1	0	1	1	0
1	1	1	1	1	0

Figure 1.1: A simple transformation diagram is shown for a transformation γ, along with its explicit definition. If τ assigns a to *true*, then $\gamma(\tau)$ assigns c the value *false*. If instead τ assigns a to *false*, then $\gamma(\tau)$ assigns b to *true* and γ does not affect c.

demonstrating the flexibility of transformations over the typical, partial assignments used for witnesses.

Transformations were defined in Chapter 3 in order to describe a general form of redundancy for Boolean functions. In the characterization presented there, transformations were used in composition with functions; in the same way a partial assignment ω can be applied to a formula F to produce $F|_\omega$, a transformation ω can be composed with a Boolean function f to produce a new function $f \circ \omega$. However, this characterization was abstract in nature, and data structures for defining transformations, or procedures for producing an expression for compositions such as $f \circ \omega$ were not discussed.

A transformation ω can certainly be defined in a manner similar to a truth table, by listing the output assignment $\omega(\tau)$ for each assignment τ. However, as for defining Boolean functions by truth tables, this is not practical if the assignments include more than a small number of variables. In this chapter we introduce transformation diagrams, which are directed acyclic graphs that define transformations. Similar to BDDs, there is a single root vertex, each non-terminal vertex is labeled by a variable, and each edge is labeled by a truth-value. In a transformation diagram, the terminal vertex has no outgoing edges, while all other vertices have either two edges with different labels, or a single edge. Vertices with a single outgoing edge are called output vertices, as they correspond to changes between input and output assignments. An example transformation diagram is shown in Figure 1.1.

We provide simple recursive procedures for working with transformation diagrams. First, for a transformation diagram G, the procedure Apply can be used to compute the image $\gamma_G(\tau)$ of any assignment τ, defined at least on the variables occurring in G, under the transformation γ_G defined by G. Second, we define a

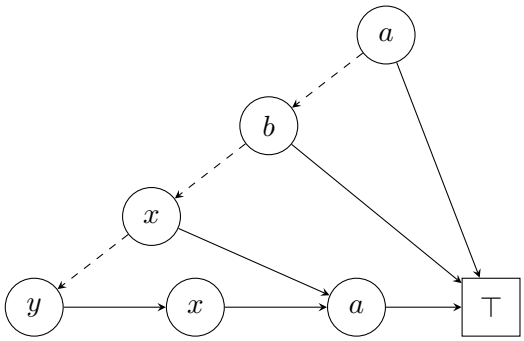


Figure 1.2: A diagram for a transformation γ which is a witness for a covered clause.

procedure Comp and prove that it computes an expression for the composition $f \circ \gamma$ given a Boolean expression f. Moreover, if the expression f is a formula F in conjunctive normal form, then the expression returned by Comp is easy to translate to conjunctive normal form as well.

We provide as Proposition 4.4.1 a simple characterization of redundancy using transformations; namely, a function g is redundant with respect to a function f if and only if there exists a transformation γ such that $f \vDash (f \wedge g) \circ \gamma$; in this case we call γ a witness for g with respect to f. Further, we show how this generalizes the typical characterization of redundant clauses using partial assignment witnesses, providing in Example 4.4.1 a construction of a graph G such that its transformation γ is a witness for a clause C with respect to a formula F, given a partial assignment witness for C with respect to F.

Finally, if C is an asymmetric covered clause with respect to a formula F, we show how to construct a diagram G, linear in the size of the extended clause C_{EXT}, such that γ is a witness for C with respect to the formula F. As an example, recall that the clause $C = (a \vee b)$ is covered with respect to

$$F = (\neg a \vee \neg b \vee c) \wedge (\neg a \vee x) \wedge (\neg c \vee \neg x \vee y) \wedge (\neg x \vee y) \wedge (\neg b \vee \neg y).$$

Figure 1.2 shows a transformation diagram which is shown to be a witness for C with respect to F in Example 4.4.2.

Chapter 5: Conclusion

In the final chapter, we briefly review the contributions made in this thesis and provide a discussion of possible directions for future work.

Chapter 2

Covered Clauses Are Not Propagation Redundant

Publication. This chapter appears in the proceedings of the 10th International Joint Conference on Automated Reasoning (IJCAR), published in 2020 by Springer as Lecture Notes in Computer Science vol. 12166 (edited by Peltier, N. and Sofronie-Stokkermans, V.), pages 32–47.

Authors. Lee A. Barnett, David Cerna, and Armin Biere

Acknowledgement. Supported by the Linz Institute of Technology AI Lab funded by the State of Upper Austria, as well as the Austrian Science Fund (FWF) under project W1255-N23, the LogiCS Doctoral College on Logical Methods in Computer Science.

Modifications. This chapter includes subsection headings, not appearing in the original publication, for added structure.

Abstract. Propositional proof systems based on recently-developed redundancy properties admit short refutations for many formulas traditionally considered hard. Redundancy properties are also used by procedures which simplify formulas in conjunctive normal form by removing redundant clauses. Revisiting the covered clause elimination procedure, we prove the correctness of an explicit algorithm for identifying covered clauses, as it has previously only been implicitly described. While other elimination procedures produce redundancy witnesses for compactly reconstructing solutions to the original formula, we prove that witnesses for covered clauses are hard to compute. Further, we show that not all covered clauses are propagation redundant, the most general, polynomially-verifiable standard redundancy property. Finally, we close a gap in the literature by demonstrating the complexity of clause redundancy itself.

2.1 Introduction

Boolean satisfiability (SAT) solvers have become successful tools for solving reasoning problems in a variety of applications, from formal verification [24] and security [74] to pure mathematics [44, 53, 65]. Significant recent progress in the design of SAT solvers has come as a result of exploiting the notion of clause redundancy (for instance, [50, 54, 55]). For a propositional formula F in conjunctive normal form (CNF), a clause C is redundant if it can be added to, or removed from, F without affecting whether F is satisfiable [64].

In particular, redundancy forms a basis for clausal proof systems. These systems refute an unsatisfiable CNF formula F by listing instructions to add or delete clauses to or from F, where the addition of a clause C is permitted only if C meets some criteria ensuring its redundancy. By eventually adding the empty clause, the formula is proven to be unsatisfiable. Crucially, the redundancy criteria of a system can also be used as an inference rule by a solver searching for such refutations, or for satisfying assignments.

Proof systems based on the recently introduced PR (Propagation Redundancy) criteria [51] have been shown to admit short refutations of the famous pigeonhole formulas [46, 52]. These are known to have only exponential-size refutations in many systems, including resolution [42] and constant-depth Frege systems [2], but have polynomial-size PR refutations. In fact, many problems typically considered hard have short PR refutations, spurring interest in these systems from the viewpoint of proof complexity [20]. Further, systems based on PR are strong even without introducing new variables, and have the potential to afford substantial improvements to SAT solvers (such as in [50, 55]).

The PR criteria is very general, encompassing nearly all other established redundancy criteria, and it is NP-complete to decide whether it is met by a given clause [55]. However, when the clause is given alongside a witness, a partial assignment providing additional evidence for the clause's redundancy, the PR criteria can be polynomially verified [51]. SAT solvers producing refutations in the PR system must find and record a witness for each PR clause addition.

Redundancy is also a basis for clause elimination procedures, which simplify a CNF formula by removing redundant clauses [47, 54]. These are useful preprocessing and inprocessing techniques that also make use of witnesses, but for the task of solution reconstruction: correcting satisfying assignments found after simplifying to ensure they solve the original formula. A witness for a clause C details how to fix assignments falsifying C without falsifying other clauses in the formula [51, 52], so solvers using elimination procedures that do not preserve formula equivalence typically provide a witness for each removed clause.

Covered clause elimination (CCE) [48] is a strong procedure which removes covered clauses, a generalization of blocked clauses [59, 67], and has been implemented in various SAT solvers (for example, [12, 13, 34]) and the CNF prepro-

cessing tool Coprocessor [70]. CCE does not preserve formula equivalence, but provides no witnesses for the clauses it removes. Instead, it uses a complex technique to reconstruct solutions in multiple steps, requiring at times a quadratic amount of space to reconstruct a single clause [54, 60]. CCE has so far only been implicitly described, and it is not clear how to produce witnesses for covered clauses.

In this paper we provide an explicit algorithm for identifying covered clauses, and show that their witnesses are difficult to produce. We also demonstrate that although covered clauses are redundant, they do not always meet the criteria required by PR. This suggests it may be beneficial to consider redundancy properties beyond PR which allow alternative types of witnesses. There has already been some work in this direction with the introduction of the SR (Substitution Redundancy) property by Buss and Thapen [20].

The paper is organized as follows. In section 2.2 we provide necessary background and terminology, while section 2.3 reviews covered clause elimination, provides the algorithm for identifying covered clauses, and proves that this algorithm and its reconstruction strategy are correct. Section 2.4 includes proofs about witnesses for covered clauses, and shows that they are not encompassed by PR. In section 2.5 we consider the complexity of deciding clause redundancy in general, followed by a conclusion and discussion of future work in section 2.6.

2.2 Preliminaries

A literal is a boolean variable x or its negation $\neg x$. A clause is a disjunction of literals, and a formula is a conjunction of clauses. We often identify a clause with the set of its literals, and a formula with the set of its clauses. For a set of literals S we write $\neg S$ to refer to the set $\neg S = \{\neg l \mid l \in S\}$. The set of variables occurring in a formula F is written $var(F)$. The empty clause is represented by $\bot$, and the satisfied clause by $\top$.

An *assignment* is a function from a set of variables to the truth values *true* and *false*. An assignment is *total* for a formula F if it assigns a value for every variable in $var(F)$, otherwise it is partial. An assignment is represented by the set of literals it assigns to true. The *composition* of assignments τ and υ is

$$\tau \circ \upsilon(x) = \begin{cases} \tau(x) & \text{if } x, \neg x \notin \upsilon \\ \upsilon(x) & \text{otherwise} \end{cases}$$

for a variable x in the domain of τ or υ. For a literal l, we write τ_l to represent the assignment $\tau \circ \{l\}$. An assignment *satisfies* (resp., *falsifies*) a variable if it assigns that variable true (resp., false). Assignments are lifted to functions assigning literals, clauses, and formulas in the usual way.

Given an assignment τ and a clause C, the *partial application* of τ to C is written $C|_\tau$ and is defined as follows: $C|_\tau = \top$ if C is satisfied by τ, otherwise, $C|_\tau = \{l \mid l \in C \text{ and } \neg l \notin \tau\}$. Likewise, the partial application of the assignment τ to a formula F is written $F|_\tau$ and defined by: $F|_\tau = \top$ if σ satisfies F, otherwise $F|_\tau = \{C|_\tau \mid C \in F \text{ and } C|_\tau \neq \top\}$. *Unit propagation* refers to the iterated application of the *unit clause rule*, replacing F by $F|_{\{l\}}$ for each unit clause $(l) \in F$, until there are no unit clauses left.

We write $F \vDash G$ to indicate that every assignment satisfying F, and which is total for G, satisfies G as well. Further, we write $F \vdash_1 G$ to mean F implies G *by unit propagation*: for every $D \in G$, unit propagation on $\neg D \wedge F$ produces $\bot$.

A clause C is *redundant* with respect to a formula F if the formulas $F \setminus \{C\}$ and $F \cup \{C\}$ are satisfiability-equivalent: both satisfiable, or both unsatisfiable [64]. The following theorem provides a characterization of clause redundancy based on logical implication.

Theorem 2.2.1 (Heule, Kiesl, and Biere [52])**.** *A non-empty clause C is redundant with respect to a formula F (with $C \notin F$) if and only if there is a partial assignment ω such that ω satisfies C, and $F|_\alpha \vDash F|_\omega$, where $\alpha = \neg C$.*

As a result, redundancy can be shown by providing a *witnessing assignment* ω (or *witness*) and demonstrating that $F|_\alpha \vDash F|_\omega$. When the logical implication relation "$\vDash$" is replaced with "$\vdash_1$," the result is the definition of a *propagation redundant* or PR *clause*, and ω is called a PR *witness* [51]. Determining whether a clause is PR with respect to a formula is NP-complete [55], but since it can be decided in polynomial time whether $F \vdash_1 G$ for arbitrary formulas F and G, it can be efficiently decided whether a given assignment is a PR witness.

A *clause elimination procedure* iteratively identifies and removes clauses satisfying a particular redundancy property from a formula, until no such clauses remain. A simple example is *subsumption elimination*, which removes any clauses $C \in F$ that are *subsumed* by another clause $D \in F$; that is, $D \subseteq C$. Subsumption elimination is *model-preserving*, as it only removes clauses C such that any assignment satisfying $F \setminus \{C\}$ also satisfies $F \cup \{C\}$.

Some clause elimination procedures are not model-preserving. *Blocked clause elimination* [59, 67] iteratively removes from a formula F any clauses C satisfying the following property: C is *blocked* by a literal $l \in C$ if for every clause $D \in F$ containing $\neg l$, there is some other literal $k \in C$ with $\neg k \in D$. For a blocked clause C, there may be assignments satisfying $F \setminus \{C\}$ which falsify $F \cup \{C\}$. However, blocked clauses are redundant, so if $F \cup \{C\}$ is unsatisfiable, then so is $F \setminus \{C\}$, thus blocked clause elimination is still *satisfiability-preserving*.

Clause elimination procedures which are not model-preserving must provide a way to *reconstruct* solutions to the original formula out of solutions to the reduced formula. Witnesses provide a convenient framework for reconstruction: if C is redundant with respect to F, and τ is a total or partial assignment

satisfying F but not C, then $\tau \circ \omega$ satisfies $F \cup \{C\}$, for any witness ω for C with respect to F [32, 52]. For reconstructing solutions after removing multiple clauses, a sequence σ of *witness-labeled clauses* $(\omega : C)$, called a *reconstruction stack*, can be maintained and used as follows [32, 60].

Definition 2.2.1. Given a sequence σ of witness-labeled clauses, the *reconstruction function* (w.r.t. σ) is defined recursively as follows, for an assignment τ: [1]

$$\mathcal{R}_\epsilon(\tau) = \tau, \qquad \mathcal{R}_{\sigma\cdot(\omega:D)}(\tau) = \begin{cases} \mathcal{R}_\sigma(\tau) & \text{if } \tau(D) = \top \\ \mathcal{R}_\sigma(\tau \circ \omega) & \text{otherwise.} \end{cases}$$

For a set of clauses S, a sequence σ of witness-labeled clauses satisfies the *reconstruction property* for S, or is a *reconstruction sequence* for S, with respect to a formula F if $\mathcal{R}_\sigma(\tau)$ satisfies $F \cup S$ for any assignment τ satisfying $F \setminus S$. As long as a witness is recorded for each clause C removed by a non-model-preserving procedure, even combinations of different clause elimination procedures can be used to simplify the same formula. Specifically, $\sigma = (\omega_1 : C_1) \cdots (\omega_n : C_n)$ is a reconstruction sequence for $\{C_1, \ldots, C_n\} \subseteq F$ if ω_i is a witness for C_i with respect to $F \setminus \{C_1, \ldots, C_i\}$, for all $1 \leq i \leq n$ [32].

The following lemma results from the fact that the reconstruction function satisfies $\mathcal{R}_{\sigma\cdot\sigma'}(\tau) = \mathcal{R}_\sigma(\mathcal{R}_{\sigma'}(\tau))$, for any sequences σ, σ' and assignment τ [32].

Lemma 2.2.1. *If σ is a reconstruction sequence for a set of clauses S with respect to $F \cup \{C\}$, and σ' is a reconstruction sequence for $\{C\}$ with respect to F, then $\sigma \cdot \sigma'$ is a reconstruction sequence for S with respect to F.*

2.3 Covered Clause Elimination

This section reviews covered clause elimination (CCE) and its asymmetric variant (ACCE), introduced by Heule, Järvisalo, and Biere [48], and presents an explicit algorithm implementing the more general ACCE procedure. The definitions as given here differ slightly from the original work, but are generally equivalent. A proof of correctness for the algorithm and its reconstruction sequence are given.

2.3.1 Covering Literals and Clauses

CCE is a clause elimination procedure which iteratively extends a clause by the addition of so-called "covered" literals. If at some point the extended clause becomes blocked, the original clause is redundant and can be eliminated. To make this precise, the set of *resolution candidates* in F of C upon l, written

[1] This improved variant over [32] is due to Christoph Scholl (3rd author of [32]).

$\mathrm{RC}(F, C, l)$, is defined as the collection of clauses in F with which C has a non-tautological resolvent upon l (where "$\otimes_l$" denotes resolution):

$$\mathrm{RC}(F, C, l) = \{C' \vee \neg l \in F \mid C' \vee \neg l \otimes_l C \not\equiv \top\}.$$

The *resolution intersection* in F of C upon l, written $\mathrm{RI}(F, C, l)$, consists of those literals occurring in each of the resolution candidates, apart from $\neg l$:

$$\mathrm{RI}(F, C, l) = \big(\bigcap \mathrm{RC}(F, C, l)\big) \setminus \{\neg l\}.$$

If $\mathrm{RI}(F, C, l) \neq \emptyset$, its literals are covered by l and can be used to extend C.

Definition 2.3.1. A literal k is *covered* by $l \in C$ with respect to F if $k \in \mathrm{RI}(F, C, l)$. A literal is covered by C if it is covered by some literal in C.

Covered literals can be added to a clause in a satisfiability-preserving manner, meaning that if the extended clause $C \cup \mathrm{RI}(F, C, l)$ is added to F, then C is redundant. In fact, C is a PR clause.

Proposition 2.3.1. *C is PR with respect to $F' = F \wedge (C \cup \mathrm{RI}(F, C, l))$ with witness $\omega = \alpha_l$, for $l \in C$ and $\alpha = \neg C$.*

Proof. Consider a clause $D|_\omega \in F'|_\omega$, for some $D \in F'$. We prove that ω is a PR witness by showing that $F'|_\alpha$ implies $D|_\omega$ by unit propagation. First, we know $l \notin D$, since otherwise $D|_\omega = \top$ would vanish in $F|_\omega$. If also $\neg l \notin D$, this means $D|_\omega = D|_\alpha$, and therefore $F'|_\alpha \vdash_1 D|_\omega$. Now, suppose $\neg l \in D$. Notice that D contains no other literal k such that $\neg k \in C$, since otherwise $D|_\omega = \top$ here as well. As a result $D \in \mathrm{RC}(F, C, l)$, so $\mathrm{RI}(F, C, l) \subset D$ and $\mathrm{RI}(F, C, l) \setminus C \subseteq D|_\omega$. Notice $\mathrm{RI}(F, C, l) \setminus C = (C \cup \mathrm{RI}(F, C, l))|_\alpha \in F'|_\alpha$, therefore $F'|_\alpha \vdash_1 D|_\omega$. □

Consequently, C is redundant with respect to $F \cup \{C'\}$ for any $C' \supseteq C$ constructed by iteratively adding covered literals to C. In other words, F and $(F \setminus \{C\}) \cup \{C'\}$ are satisfiability-equivalent, so that C could be replaced by C' in F without affecting the satisfiability of the formula. Thus if some such extension C' would be blocked in F, that C' would be redundant, and therefore C is redundant itself. CCE identifies and removes such clauses from F.

Definition 2.3.2. A clause C is *covered* in F if an extension of C by iteratively adding covered literals is blocked.

CCE refers to the following procedure: while some clause C in F is covered, remove C (that is, replace F with $F \setminus \{C\}$).

ACCE strengthens this procedure by extending clauses using a combination of covered literals and asymmetric literals. A literal k is *asymmetric* to C with respect to F if there is a clause $C' \vee \neg k \in F$ such that $C' \subseteq C$. The addition of

asymmetric literals to a clause is model-preserving, so that the formulas F and $(F \setminus \{C\}) \cup \{C \vee k\}$ are equivalent, for any k which is asymmetric to C [47].

Definition 2.3.3. A clause $C' \supseteq C$ is an *ACC extension* of C with respect to F if C' can be constructed from C by the iterative addition of covered and asymmetric literals. If some ACC extension of C is blocked or subsumed in F, then C is an *asymmetric covered clause* (ACC).

ACCE performs the following: while some C in F is an ACC, remove C from F. Solvers aiming to eliminate covered clauses more often implement ACCE than plain CCE, since asymmetric literals can easily be found by unit propagation, and ACCE is more powerful than CCE, eliminating more clauses [47, 48].

2.3.2 The ACC Algorithm

The procedure $\mathsf{ACC}(F, C)$ in Fig. 2.1 provides an algorithm identifying whether a clause C is an ACC with respect to a formula F. This procedure differs in some ways, and includes optimizations over the original procedure as implicitly given by the definition of ACCE. Notably, two extensions of the original clause C are maintained: E consists of C and any added covered literals, while α tracks C and all added literals, both covered and asymmetric. The literals in α are kept negated, so that $E \subseteq \neg\alpha$, and the clause represented by $\neg\alpha$ is the ACC extension of the original clause C being computed.

The E and α extensions are maintained separately for two purposes. First, the covered literal addition loop (lines 9–16) needs to iterate only over those literals in E, and can ignore those in $(\neg\alpha) \setminus E$, as argued below.

Lemma 2.3.1. *If k is covered by $l \in (\neg\alpha) \setminus E$, then $k \in \neg\alpha$ already.*

Proof. If l belongs to $\neg\alpha$ but not to E, then there is some clause $D \vee \neg l$ in F such that $D \subseteq \neg\alpha$. But then $D \vee \neg l$ occurs in $\mathrm{RC}(F, \neg\alpha, l)$, and consequently $\mathrm{RI}(F, \neg\alpha, l) \subseteq D \subseteq \neg\alpha$. Thus $k \in \mathrm{RI}(F, \neg\alpha, l)$ implies $k \in \neg\alpha$. □

Notice that the computation of the literals covered by $l \in E$ also prevents any of these literals already in $\neg\alpha$ from being added again.

The second reason for separating E and α is as follows. When a covered literal is found, or when the extended clause is blocked, the algorithm appends a new witness-labeled clause to the reconstruction sequence σ (lines 11 and 14). Instead of $(\neg\alpha_l : \alpha)$, the procedure adds the shorter witness-labeled clause $(\neg E_l : E)$. The proof of statement (3) in lemma 2.3.2 below shows that this is sufficient.

Certain details are omitted, especially concerning the addition of asymmetric literals (lines 6–7), but notice that it is never necessary to recompute $F|_\alpha$ entirely. Instead the assignment falsifying each u newly added to α can simply be applied to the existing $F|_\alpha$. In contrast, the **for each** loop (lines 9–16) should re-iterate over the entirety of E each time, as added literals may allow new coverings:

```
   ACC(F, C)
1    σ := ε
2    E :=  C
3    α := ¬C
4    repeat
5      if ⊥ ∈ F|α then return (true, σ)
6      if there are unit clauses in F|α then
7        α := α ∪ {u} for each unit u
8      else
9        for each l ∈ E
10         G := {D|α | (D ∨ ¬l) ∈ F and D|α ≠ ⊤}
11         if G = ∅ then return (true, σ · (¬E_l : E))
12         Φ := ⋂G
13         if Φ ≠ ∅ then
14           σ := σ · (¬E_l : E)
15           E := E ∪  Φ
16           α := α ∪ ¬Φ
17   until no updates to α
18   return (false, ε)
```

Figure 2.1: Asymmetric Covered Clause (ACC) Identification. The procedure $\mathsf{ACC}(F, C)$ maintains a sequence σ of witness-labeled clauses, and two sets of literals E and α. The main loop iteratively searches for literals which could be used to extend C and adds their negations to α, so that the clause represented by $\neg\alpha$ is an ACC extension of C. The set E records only those which could be added as covered literals. If C is an ACC, then $\mathsf{ACC}(F, C)$ returns $(\mathbf{true}, \sigma)$: in line 5 if the extension $\neg\alpha$ becomes subsumed in F, or in line 11 if it becomes blocked. In either case, the witness-labeled clauses in σ form a reconstruction sequence for the clause C. Note that lines 5–7 implement Boolean constraint propagation (over the partial assignment α) and can make use of efficient watched clause data structures, while line 10 has to collect all clauses containing $\neg l$, which are still unsatisfied by α, and thus requires full occurrence lists.

Example 2.3.1. Let $C = a \vee b \vee c$ and

$$F = (\neg a \vee \neg x_1) \wedge (\neg a \vee x_2) \wedge (\neg b \vee \neg x_1) \wedge (\neg b \vee \neg x_2) \wedge (\neg c \vee x_1)$$

Initially, neither a nor b cover any literals, but c covers x_1, so it can be added to the clause. After extending, a in $C \vee x_1$ covers x_2, and b blocks $C \vee x_1 \vee x_2$.

2.3.3 Correctness of ACC

The following lemma supplies invariants for arguing about $\mathsf{ACC}(F, C)$.

Lemma 2.3.2. *After each update to α, for the clauses represented by $\neg\alpha$ and E:*

(1) *$\neg\alpha$ is an* ACC *extension of C,*

(2) *$F \cup \{\neg\alpha\} \vDash F \cup \{E\}$, and*

(3) *σ is a reconstruction sequence for $\{C\}$ with respect to $F \cup \{\neg\alpha\}$.*

Proof. Let α_i, σ_i, and E_i refer to the values of α, σ, and E, respectively, after $i \geq 0$ updates to α (so that $\alpha_i \subsetneq \alpha_{i+1}$ for each i, but possibly $\sigma_i = \sigma_{i+1}$ and $E_i = E_{i+1}$). Initially, (1) and (2) hold as $E = \neg\alpha_0 = C$. Further, $\sigma_0 = \epsilon$ is a reconstruction sequence for $\{C\}$ with respect to $F \cup \{C\}$, so (3) holds as well. Assuming these claims hold after update i, we show that they hold after $i + 1$.

First suppose update $i + 1$ is the result of executing line 7.

(1) $\alpha_{i+1} = \alpha_i \cup U$, where $u \in U$ implies (u) is a unit clause in $F|_{\alpha_i}$. Then $\neg\alpha_{i+1}$ is the extension of $\neg\alpha_i$ by the addition of asymmetric literals $\neg U$. Assuming $\neg\alpha_i$ is an ACC extension of C, then so is $\neg\alpha_{i+1}$.

(2) Asymmetric literal addition is model-preserving, so $F \cup \{\neg\alpha_{i+1}\} \vDash F \cup \{\neg\alpha_i\}$. Since E was not updated, $E_{i+1} = E_i$. Assuming $F \cup \{\neg\alpha_i\} \vDash F \cup \{E_i\}$, we get $F \cup \{\neg\alpha_{i+1}\} \vDash F \cup \{E_{i+1}\}$.

(3) Again, asymmetric literal addition is model-preserving. Assuming σ_i is a reconstruction sequence for $\{C\}$ with respect to $F \cup \{\alpha_i\}$, then lemma 2.2.1 implies $\sigma_{i+1} = \sigma_i \cdot \varepsilon = \sigma_i$ reconstructs $\{C\}$ with respect to $F \cup \{\neg\alpha_{i+1}\}$.

Now, suppose instead update $i + 1$ is executed in line 16.

(1) $\alpha_{i+1} = \alpha_i \cup \Phi$, for some set of literals $\Phi \neq \emptyset$ constructed for $l \in E \subseteq \neg\alpha$. Notice for $k \in \Phi$ that $k \in \mathrm{RI}(F, \neg\alpha, l)$, so k is covered by $\neg\alpha$. Thus assuming $\neg\alpha_i$ is a ACC extension of C, then $\neg\alpha_{i+1}$ is as well.

(2) Consider an assignment τ satisfying $F \cup \{\neg\alpha_{i+1}\}$. If τ satisfies $\neg\alpha_i \subset \neg\alpha_{i+1}$ then τ satisfies $F \cup \{\neg\alpha_i\}$ and by assumption, $F \cup \{E_i\}$. Since $E_i \subset E_{i+1}$ in this case, τ satisfies $F \cup \{E_{i+1}\}$. If instead τ satisfies some literal in $\neg\alpha_{i+1} \setminus \neg\alpha_i$ then τ satisfies $\Phi \subseteq E_{i+1}$, so τ satisfies $F \cup \{E_{i+1}\}$. Thus $F \cup \{\neg\alpha_{i+1}\} \models F \cup \{E_{i+1}\}$ in this case as well.

(3) Proposition 2.3.1 implies $((\alpha_i)_l : \neg\alpha_i)$ is a reconstruction sequence for $\{\neg\alpha_i\}$ with respect to $F \cup \{\neg\alpha_{i+1}\}$. As $E_i \subseteq \neg\alpha_i$, and $F \cup \{\neg\alpha_i\} \models F \cup \{E_i\}$ by assumption, then any τ falsifies $\neg\alpha_i$ if and only if τ falsifies E_i. Since $l \in E_i$ as well, then $((\neg E_i)_l : E_i)$ is, in fact, also a reconstruction sequence for $\{\neg\alpha_i\}$ with respect to $F \cup \{\neg\alpha_{i+1}\}$. Finally, with the assumption σ_i is a reconstruction sequence for C with respect to $F \cup \{\neg\alpha_i\}$ and lemma 2.2.1, then $\sigma_{i+1} = \sigma_i \cdot ((\neg E_i)_l : E_i)$ is a reconstruction sequence for $\{C\}$ in $F \cup \{\neg\alpha_{i+1}\}$.

Thus both updates maintain invariants (1)–(3). □

With the help of this lemma we can now show the correctness of $\mathsf{ACC}(F, C)$:

Theorem 2.3.1. *For a formula F and a clause C, the procedure $\mathsf{ACC}(F, C)$ returns $(\textbf{\textit{true}}, \sigma)$ if and only if C is an* ACC *with respect to F. Further, if $\mathsf{ACC}(F, C)$ returns $(\textbf{\textit{true}}, \sigma)$, then σ is a reconstruction sequence for $\{C\}$ with respect to F.*

Proof. ($\Rightarrow$) Suppose $(\textbf{true}, \sigma)$ is returned in line 5. Then $\bot \in F|_\alpha$, so there is some $D \in F$ such that $D \subseteq \neg\alpha$; that is, $\neg\alpha$ is subsumed by D. By lemma 2.3.2 then an ACC extension of C is subsumed in F, so C is an ACC with respect to F. Further, subsumption elimination is model-preserving, so that lemmas 2.2.1 and 2.3.2 imply σ is a reconstruction sequence for C with respect to F.

Suppose now that $(\textbf{true}, \sigma)$ is returned in line 11. Then for α and some $l \in E$, all clauses in F with $\neg l$ are satisfied by α. Since $E \subseteq \neg\alpha$, then $\neg\alpha$ is blocked by l. By lemma 2.3.2 then C is an ACC with respect to F. Now, α_l is a witness for $\neg\alpha$ with respect to F, and $(\alpha_l : \neg\alpha)$ is a reconstruction sequence for $\{\neg\alpha\}$ in F. Further, $E \subseteq \neg\alpha$, and lemma 2.3.2 gives $F \cup \{\neg\alpha\} \models F \cup \{E\}$, therefore $((\neg E_i)_l : E_i)$ is a reconstruction sequence for $\{\neg\alpha\}$ in F as well. Then lemma 2.2.1 implies $\sigma \cdot (\neg E_l : E)$ is a reconstruction sequence for C with respect to F.

($\Leftarrow$) Suppose C is an ACC; that is, some $C' = C \vee k_1 \vee \cdots \vee k_n$ is blocked or subsumed in F, where k_1 is an asymmetric or covered literal for C, and k_i is an asymmetric or covered literal for $C \vee k_1 \vee \cdots \vee k_{i-1}$ for $i > 1$. Towards a contradiction, assume $\mathsf{ACC}(F, C)$ returns $(\textbf{false}, \varepsilon)$. Then for the final value of α, the clause represented by $\neg\alpha$ is not blocked nor subsumed in F, and hence, $C' \not\subseteq \neg\alpha$. As $C \subseteq \neg\alpha$, there must be some values of i such that $\neg k_i \notin \alpha$.

Let m refer to the least such i; that is, $\neg k_m \notin \alpha$, but $\neg k_i \in \alpha$ for all $1 \leq i < m$. Thus k_m is asymmetric, or covered by, $C_{m-1} = C \vee k_1 \vee \cdots \vee k_{m-1}$.

If k_m is asymmetric to C_{m-1}, there is some clause $D \vee \neg k_m$ in F such that $D \subseteq C_{m-1}$. By assumption, $\neg k_m \notin \alpha$ but $\neg C_{m-1} \subseteq \alpha$. Further, $k_m \notin \alpha$, as otherwise $(D \vee \neg k_m)|_\alpha = \bot$ and $\mathsf{ACC}(F, C)$ would have returned **true**. But $(D \vee \neg k_m)|_\alpha = \neg k_m$ would be a unit in $F|_\alpha$ and added to α by line 7.

If instead k_m is covered by C_{m-1}, then $k_m \in \mathrm{RI}(F, C_{m-1}, l)$ for some literal $l \in C_{m-1} \subseteq \neg\alpha$. In fact $l \in E$, by lemma 2.3.1. During the l^{th} iteration of the **for each** loop, then $k_m \in \Phi$, and $\neg k_m$ would be added to α by line 16.

□

2.3.4 Solution Reconstruction

$\mathsf{ACC}(F, C)$ produces, for any asymmetric covered clause C in F, a reconstruction sequence σ for C with respect to F. This allows ACCE to be used during preprocessing or inprocessing like other clause elimination procedures, appending this σ to the solver's main reconstruction stack whenever an ACC is removed. However, the algorithm does not produce redundancy witnesses for the clauses it removes. Instead, σ consists of possibly many witness-labeled clauses, starting with the redundant clause C, and reconstructs solutions for C in multiple steps.

In contrast, most clause elimination procedures produce a single witness-labeled clause $(\omega : C)$ for each removed clause C. In practice, only the part of ω which differs from $\neg C$ must be recorded; for most procedures this difference includes only literals in C, so that reconstruction for $\{C\}$ needs only linear space in the size of C. In contrast, the size of σ produced by $\mathsf{ACC}(F, C)$ to reconstruct $\{C\}$ can be quadratic in the length of the extended clause.

Example 2.3.2. Consider $C = x_0$ and

$$F_n = (\neg x_{n-2} \vee x_{n-1} \vee x_n) \wedge (\neg x_{n-1} \vee \neg x_n) \wedge \bigwedge_{i=1}^{n-2} (\neg x_{i-1} \vee x_i).$$

The extended clause $\neg\alpha = x_0 \vee x_1 \vee \cdots \vee x_n$ is blocked in F_n by x_{n-1}. Then $\mathsf{ACC}(F_n, C)$ returns the pair with **true** and the reconstruction sequence[2]

$$\sigma = (x_0 \wr x_0) \cdot (x_1 \wr x_0 \vee x_1) \cdots (x_{n-2} \wr x_0 \vee x_1 \vee \cdots \vee x_{n-2}) \cdot (x_{n-1} \wr x_0 \vee x_1 \vee \cdots \vee x_n).$$

The extended clause includes n literals, and the size of σ is $O(n^2)$.

[2] In order to simplify the presentation, only the part of the witness differing from the negated clause is written, so that $(l \wr C)$ actually stands for $(\neg C_l : C)$. The former is in essence the original notation used in [60], while set, super or globally blocked, as well as PR clauses [51, 63, 64] require the more general one used in this paper.

2.4 Witnesses for Covered Clauses

In this section, we consider the specific problem of finding witnesses for (asymmetric) covered clauses. As these clauses are redundant, such witnesses are guaranteed to exist by theorem 2.2.1, though they are not produced by $\mathsf{ACC}(F, C)$. More precisely, we are interested in the *witness problem* for covered clauses.

2.4.1 The Witness Problem

Definition 2.4.1. The *witness problem* for a redundancy property P is as follows: given a formula F and a clause C, if P is met by C with respect to F then return a witness for C, or decide that P is not met by C.

For instance, the witness problem for blocked clauses is solved as follows: test each $l \in C$ to see if l blocks C in F. As soon as a blocking literal l is found then α_l is a witness for C, where $\alpha = \neg C$. If no blocking literal is found, then C is not blocked. For blocked clauses, this polynomial procedure decides whether C is blocked or not and also determines a witness $\omega = \alpha_l$ for C.

Solving the witness problem for covered clauses is not as straightforward, as it is not clear how a witness could be produced when deciding a clause is covered, or from a sequence σ constructed by $\mathsf{ACC}(F, C)$. The following theorem shows that this problem is as difficult as producing a satisfying assignment for an arbitrary formula, if one exists. In particular, we present a polynomial time reduction from the search analog of the SAT problem: given a formula F, return a satisfying assignment of F, or decide that F is unsatisfiable.

Specifically, given a formula G, we construct a pair (F, C) as an instance to the witness problem for covered clauses. In this construction, C is covered in F and has some witness ω. Moreover, any witness ω for this C necessarily provides a satisfying assignment to G, if there is one.

Proposition 2.4.1. *Given a formula $G = D_1 \wedge \cdots \wedge D_n$, let $G' = D'_1 \wedge \cdots \wedge D'_n$ refer to a variable-renamed copy of G, containing v' everywhere G contains v, so that $var(G) \cap var(G') = \emptyset$. Further, let $C = k \vee l$ and construct the formula:*

$$\begin{array}{rcl} F \;=\; & (x \vee \neg k) \wedge (\neg x \;\vee \neg y) \wedge (y \vee \neg l) & \wedge \\ & (x \vee D_1) \wedge \quad \cdots \quad \wedge (x \vee D_n) & \wedge \\ & (y \vee D'_1) \wedge \quad \cdots \quad \wedge (y \vee D'_n) & \end{array}$$

for variables $x, y, k, l \notin var(G) \cup var(G)'$. Finally, let ω be a witness for C with respect to F. Either ω satisfies at least one of G or G', or G is unsatisfiable.

Proof. First notice for C that x is covered by k and y is covered by l, so that the extension $(k \vee l \vee x \vee y)$ is blocked in F (with blocking literal x or y). Thus C is redundant in F, so a witness ω exists.

We show that ω satisfies G or G' if and only if G is satisfiable.

($\Rightarrow$) If ω satisfies G then surely G is satisfiable. If ω satisfies G' but not G then the assignment $\omega_G = \{x \in var(G) \mid x' \in G' \text{ and } x \in \omega\}$ satisfies G.

($\Leftarrow$) Assume G is satisfiable, and without loss of generality[3] further assume $\omega = \{k\} \circ \omega'$ for some ω' not assigning $var(k)$. Then $F|_\alpha \vDash (F|_{\{k\}})|_{\omega'}$; that is,

$$F|_\alpha \vDash \big((x) \wedge (\neg x \vee \neg y) \wedge (y \vee \neg l) \wedge (x \vee D_1) \wedge \cdots \wedge (y \vee D'_n)\big)|_{\omega'}.$$

G is satisfiable, so there are models of $F|_\alpha$ in which $\neg x$ is true. However, x occurs as a unit clause in $F|_{\{k\}}$, so it must be the case that $x \in \omega'$. Therefore $\omega = \{k, x\} \circ \omega''$ for some ω'' assigning neither $var(k)$ nor $var(x)$ such that

$$F|_\alpha \vDash \big((\neg y) \wedge (y \vee \neg l) \wedge (y \vee D'_1) \wedge \cdots \wedge (y \wedge D'_n)\big)|_{\omega''}.$$

By similar reasoning, ω'' must assign y to false, so now $\omega = \{k, x, \neg y\} \circ \omega'''$ for some ω''', assigning none of $var(k)$, $var(x)$, or $var(y)$, such that

$$F|_\alpha \vDash \big((\neg l) \wedge (D'_1) \wedge \cdots \wedge (D'_n)\big)|_{\omega'''}.$$

Finally, consider any clause $D'_i \in G'$. We show that ω satisfies D'_i. As ω is a witness, $F|_\alpha \vDash F|_\omega$, so that $(D'_i)|_\omega$ is true in all models of $F|_\alpha$, including models which assign y to true. In particular, let τ be a model of G; then $(D'_i)|_\omega$ is satisfied by $\tau \cup \{\neg x, y\} \cup \nu$, for every assignment ν over $var(G')$. Because $var(D'_i) \subseteq var(G')$, then $(D'_i)|_\omega \equiv \top$. Therefore $G'|_\omega \equiv \top$.

□

Proposition 2.4.1 suggests there is likely no polynomial procedure for computing witnesses for covered clauses. The existence of witnesses is the basis for solution reconstruction, but witnesses which cannot be efficiently computed make the use of non-model-preserving clause elimination procedures more challenging; that is, we are not aware of any polynomial algorithm for generating a compact (sub-quadratic) reconstruction sequence (see also example 2.3.2).

2.4.2 Covered Clauses And PR

As PR clauses are defined by witnesses, procedures deciding PR generally solve the witness problem for PR. For example, the PR reduct [50] provides a formula whose satisfying assignments encode PR witnesses, if they exist. However, this does not produce witnesses for covered clauses, which are not encompassed by PR. In other words, although any clause extended by a single covered literal addition is a PR clause by proposition 2.3.1, this is not true for covered clauses.

[3] If $k \notin \omega$, then $\omega = \{l\} \circ \omega'$ for some ω' not assigning $var(l)$ and the argument is symmetric, ending with $G|_\omega = \top$. Note that by definition ω satisfies C thus k or l.

Theorem 2.4.1. *Covered clauses are not all propagation redundant.*

Proof. By counterexample. Consider the clause $C = k \vee l$ and the formula

$$\begin{aligned} F \;=\; & (x \vee \neg k) \;\wedge\; (\neg x \;\vee\; \neg y) \;\wedge (y \vee \neg l) \;\wedge \\ & (x \vee a \vee b) \wedge (x \vee a \vee \neg b) \;\wedge\; (x \vee \neg a \vee b) \wedge (x \vee \neg a \vee \neg b) \;\wedge \\ & (y \vee c \vee d) \wedge (y \vee c \vee \neg d) \;\wedge\; (y \vee \neg c \vee d) \wedge (y \vee \neg c \vee \neg d). \end{aligned}$$

The extension $C \vee x \vee y$ is blocked with respect to F, so C is covered. However, C is not PR with respect to F. To see this, suppose to the contrary that ω is a PR witness for C. Similar to the reasoning in the proof of theorem 2.4.1, assume, without loss of generality, that $\omega = \{k\} \circ \omega'$ for some ω' not assigning k. Notice that $(x) \in F|_k$, but unit propagation on $\neg x \wedge F|_\alpha$ stops without producing $\bot$. Therefore $x \in \omega'$, and $\omega = \{k, x\} \circ \omega''$ for some ω'' assigning neither k nor x. By similar reasoning, it must be the case that $\neg y \in \omega''$, so that $\omega = \{k, x, \neg y\} \circ \omega'''$. Now, $(c \vee d) \in F|_{\{k,x,\neg y\}}$, but once more, unit propagation on $F|_\alpha \wedge \neg c \wedge \neg d$ does not produce $\bot$, so either c or d belongs to ω'''. Without loss of generality, assume $c \in \omega'''$ so that $\omega = \{k, x, \neg y, c\} \circ \omega''''$. Finally, both d and $\neg d$ are clauses in $F'|_{\{k,x,\neg y,c\}}$, but neither are implied by $F|_\alpha$ by unit propagation. However, if either d or $\neg d$ belongs to ω, then $\bot \in F|_\omega$. As unit propagation on $F|_\alpha$ alone does not produce $\bot$, this is a contradiction. □

Notice the formula in theorem 2.4.1 can be seen as an instance of the formula in proposition 2.4.1, with G as $(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b)$. In fact, as long as unit propagation on G does not derive $\bot$, then G could be any, arbitrarily hard, unsatisfiable formula (such as an instance of the pigeonhole principle).

2.5 Complexity of Redundancy

In the previous section we introduced the witness problem for a redundancy property (definition 2.4.1) and showed that it is not trivial, even when the redundancy property itself can be efficiently decided. Further, the witness problem for PR clauses is solvable by encoding it into SAT [50].

Note that PR is considered to be a very general redundancy property. The proof of theorem 1 in [51, 52] shows that if F is satisfiable and C redundant, then $F \wedge C$ is satisfiable by definition. In addition, any satisfying assignment τ of $F \wedge C$ is a PR witness for C with respect to F. This yields the following:

Proposition 2.5.1. *Let F be a satisfiable formula. A clause C is redundant with respect to F if and only if it is a* PR *clause with respect to F.*

While not all covered clauses are PR, this motivates the question of whether witnesses for all redundancy properties can be encoded as an instance to SAT,

and solved similarly. In this section we show that this is likely not the case by demonstrating the complexity of the *redundancy problem*: given a clause C and a formula F, is C redundant with respect to F?

Deciding whether a clause is PR belongs to NP: assignments can be chosen non-deterministically and efficiently verified as PR witnesses, since the relation $\vdash_1$ is polynomially decidable [55]. For clause redundancy in general, it is not clear that this holds, as the corresponding problem is co-NP-complete.

Proposition 2.5.2. *Deciding whether an assignment ω is a witness for a clause C with respect to a formula F is complete for* co-NP*.*

Proof. The problem belongs to co-NP since $F|_\alpha \vDash F|_\omega$ whenever $\neg(F|_\alpha) \vee F|_\omega$ is a tautology. In the following we show a reduction from the tautology problem. Given a formula F, construct the formula F' as below, for $x \notin var(F)$. Further, let $C' = x$, so that $\alpha = \neg x$, and let also $\omega = x$.

$$F' = \bigwedge_{C \in F} (C \vee \neg x)$$

Then $F'|_\alpha = \top$ and $F'|_\omega = F$. Therefore $F'|_\alpha \vDash F'|_\omega$ if and only if $\top \vDash F$. □

Theorem 2.5.1 below shows that the *irredundancy problem*, the complement of the redundancy problem, is complete for the class $\mathrm{D^P}$, the class of languages that are the intersection of a language in NP and a language in co-NP [79]:

$$\mathrm{D^P} = \{L_1 \cap L_2 \mid L_1 \in \mathrm{NP} \text{ and } L_2 \in \text{co-NP}\}.$$

This class was originally introduced to classify certain problems which are hard for both NP and co-NP, but do not seem to be complete for either, and it characterizes a variety of optimization problems. It is the second level of the Boolean hierarchy over NP, which is the completion of NP under Boolean operations [21, 98]. We provide a reduction from the canonical $\mathrm{D^P}$-complete problem, SAT-UNSAT: given formulas F and G, is F satisfiable and G unsatisfiable?

Theorem 2.5.1. *The irredundancy problem is* $\mathrm{D^P}$*-complete.*

Proof. Notice that the irredundancy problem can be expressed as

$$\begin{aligned}\mathrm{IRR} &= \{\,(F, C) \mid F \text{ is satisfiable, and } F \wedge C \text{ is unsatisfiable}\} \\ &= \{\,(F, C) \mid F \in \mathrm{SAT}\} \cap \{(F, C) \mid F \wedge C \in \mathrm{UNSAT}\}.\end{aligned}$$

That is, IRR is the intersection of a language in NP and a language in co-NP, and so the irredundancy problem belongs to $\mathrm{D^P}$.

Now, let (F, G) be an instance to SAT-UNSAT. Construct the formula F' as follows, for $x \notin var(F) \cup var(G)$:

$$F' = \bigwedge_{C \in F} (C \vee x) \wedge \bigwedge_{D \in G} (D \vee \neg x).$$

Further, let $C' = x$. We demonstrate that $(F, G) \in$ SAT-UNSAT if and only if C' is irredundant with respect to F'.

($\Leftarrow$) Suppose C' is irredundant with respect to F'. In other words, F' is satisfiable but $F' \wedge C'$ is unsatisfiable. Since $F' \wedge C'$ is unsatisfiable, it must be the case that $F'|_{\{x\}}$ is unsatisfiable; however, F' is satisfiable, therefore $F'|_{\{\neg x\}}$ must be satisfiable. Since $F'|_{\{\neg x\}} = F$ and $F'|_{\{x\}} = G$, then $(F, G) \in$ SAT-UNSAT.

($\Rightarrow$) Now, suppose F is satisfiable and G is unsatisfiable. Then some assignment τ over $var(F)$ satisfies F. As a result, $\tau \cup \{\neg x\}$ satisfies F'. Because G is unsatisfiable, there is no assignment satisfying $F'|_{\{x\}} = G$. This means there is no σ satisfying both F' and $C' = x$, and so $F' \wedge C'$ is unsatisfiable as well. Therefore C' is irredundant with respect to F'.

□

Consequently the redundancy problem is complete for co-$\mathrm{D^P}$. This suggests that sufficient SAT encodings of the clause redundancy problem, and its corresponding witness problem, are not possible.

2.6 Conclusion

We revisit a strong clause elimination procedure, covered clause elimination, and provide an explicit algorithm for both deciding its redundancy property and reconstructing solutions after its use. Covered clause elimination is unique in that it does not produce redundancy witnesses for clauses it eliminates, and uses a complex, multi-step reconstruction strategy. We prove that while witnesses exist for covered clauses, computing such a witness is as hard as finding a satisfying assignment for an arbitrary formula.

For PR, a very general redundancy property used by strong proof systems, witnesses can be found through encodings into SAT. We show that covered clauses are not described by PR, and SAT encodings for finding general redundancy witnesses likely do not exist, as deciding clause redundancy is hard for the class $\mathrm{D^P}$, the second level of the Boolean hierarchy over NP.

Directions for future work include the development of redundancy properties beyond PR, and investigating their use for solution reconstruction after clause elimination, as well as in proof systems. Extending redundancy notions by using a structure for witnesses other than partial assignments may provide more generality while remaining polynomially verifiable.

We are also interested in developing notions of redundancy for adding or removing more than a single clause at a time, and exploring proof systems and simplification techniques which make use of non-clausal redundancy properties.

Chapter 3

Non-Clausal Redundancy Properties

Publication. This chapter appears in the proceedings of the 28th International Conference on Automated Deduction (CADE), published in 2021 by Springer as Lecture Notes in Computer Science vol. 12699 (edited by Platzer, A. and Sutcliffe, G.), pages 252–272. It is also Technical Report 21/2 in the FMV Reports Series at Johannes Kepler University.

Authors. Lee A. Barnett and Armin Biere

Acknowledgement. Supported by the Linz Institute of Technology AI Lab funded by the State of Upper Austria, as well as the Austrian Science Fund (FWF) under project W1255-N23, the LogiCS Doctoral College on Logical Methods in Computer Science.

Modifications. The appendix included at the end of this chapter does not appear in the CADE proceedings. This chapter also includes subsection headings, not appearing in earlier publications, for added structure.

Abstract. State-of-the-art refutation systems for SAT are largely based on the derivation of clauses meeting some redundancy criteria, ensuring their addition to a formula does not alter its satisfiability. However, there are strong propositional reasoning techniques whose inferences are not easily expressed in such systems. This paper extends the redundancy framework beyond clauses to characterize redundancy for Boolean constraints in general. We show this characterization can be instantiated to develop efficiently checkable refutation systems using redundancy properties for Binary Decision Diagrams (BDDs). Using a form of reverse unit propagation over conjunctions of BDDs, these systems capture, for instance, Gaussian elimination reasoning over XOR constraints encoded in a formula, without the need for clausal translations or extension variables. Notably, these systems generalize those based on the strong Propagation Redundancy (PR) property, without an increase in complexity.

3.1 Introduction

The correctness and reliability of Boolean satisfiability (SAT) solvers is critical for many applications. For instance SAT solvers are used for verifying hardware and software systems (e.g. [24, 40, 61]), to search for solutions to open problems in mathematics (e.g. [53, 65]), and as subroutines of other logical reasoning tools (e.g. [9, 96]). Solvers should be able to provide solution certificates that are easily and externally checkable. For a satisfiable formula, any satisfying assignment is a suitable certificate and typically can be easily produced by a solver. For an unsatisfiable formula, a solver should be able to produce a refutation proof.

Modern SAT solvers primarily refute unsatisfiable formulas using clausal proof systems, such as the popular DRAT system [99] used by the annual SAT competition in recent years [6], or newer systems based on the surprisingly strong Propagation Redundancy (PR) property [51]. Clausal proof systems iteratively extend a formula, typically given in conjunctive normal form (CNF), by adding clauses that are redundant; that is, their addition to the formula does not affect whether it is satisfiable. Systems are distinguished by their underlying redundancy properties, restricted but efficiently-decidable forms of redundancy.

Redundancy is a useful notion in SAT as it captures most inferences made by state-of-the-art solvers. This includes clauses implied by the current formula, such as the resolvent of two clauses or clauses learned during conflict-driven clause learning (CDCL) [10, 72], as well as clauses which are not implied but derived nonetheless by certain preprocessing and inprocessing techniques [60], such as those based on blocked clauses [59, 64, 67]. Further, clausal proof systems based on properties like PR include short refutations for several hard families of formulas, such as those encoding the pigeonhole principle, that have no polynomial-length refutations in resolution [2] (see [20] for an overview). These redundancy properties, seen as inference systems, thus potentially offer significant improvements in efficiency, as the CDCL algorithm at the core of most solvers searches only for refutations in resolution [11]. While the recent satisfaction-driven clause learning (SDCL) paradigm has shown some initial success [50, 55], it is still unclear how to design solving techniques which take full advantage of this potential.

Conversely, there are existing strong reasoning techniques which similarly exceed the abilities of CDCL alone, but are difficult to express using clausal proof systems. Important examples include procedures for reasoning over CNF formulas encoding pseudo-Boolean and cardinality constraints (see [84]), as well as Gaussian elimination (see [15, 88, 89, 97]), which has been highlighted as a challenge for clausal proof systems [45]. Gaussian elimination, applied to sets of "exclusive-or" (XOR) constraints, is a crucial technique for many problems from cryptographic applications [89], and can efficiently solve, for example, Tseitin formulas hard for resolution [92, 94]. This procedure, implemented by CryptoMin-

iSAT [89], Lingeling [13], and Coprocessor [70] for example, can be polynomially simulated by extended resolution, allowing inferences over new variables, and similar systems (see [80, 86]). However due to the difficulty of such simulations they are not typically implemented. Instead solvers supporting these techniques simply prevent them from running when proof output is required, preferring less efficient techniques whose inferences can be more easily represented.

This paper extends the redundancy framework for clausal proof systems to include non-clausal constraints, such as XOR or cardinality constraints, presenting a characterization of redundancy for Boolean functions in general. We demonstrate a particular use of this characterization by instantiating it for functions represented by Binary Decision Diagrams [17], a powerful representation with a long history in SAT solving (e.g. [18, 29, 33, 76, 78]) and other areas of automated reasoning (e.g. [19, 41, 66, 81]). We show the resulting refutation systems succinctly express Gaussian elimination while also generalizing existing clausal systems. Results using a prototype implementation confirm these systems allow compact and efficiently checkable refutations of CNF formulas that include embedded XOR constraints solvable by Gaussian elimination.

In the rest of the paper, Section 3.2 includes preliminaries and Section 3.3 presents the characterization of redundancy for Boolean functions. Section 3.4 introduces redundancy properties for BDDs, and Section 3.5 demonstrates their use for Gaussian elimination. Section 3.6 presents the results of our preliminary implementation, and Section 3.7 concludes.

3.2 Preliminaries

We assume a set of Boolean variables V under a fixed order $\prec$ and use standard SAT terminology. The set of truth values is $B = \{0, 1\}$. An *assignment* is a function $\tau : V \to B$ and the set of assignments is B^V. A function $f : B^V \to B$ is *Boolean*. If $f(\tau) = 1$ for some $\tau \in B^V$ then f is *satisfiable*, otherwise f is *unsatisfiable*. Formulas express Boolean functions as usual, are assumed to be in conjunctive normal form, and are written using capital letters F and G. A clause can be represented by its set of literals and a formula by its set of clauses.

A *partial assignment* is a non-contradictory set of literals σ; that is, if $l \in \sigma$ then $\neg l \notin \sigma$. The *application* of a partial assignment σ to a clause C is written $C|_\sigma$ and defined by: $C|_\sigma = \top$ if every $\tau \in B^V$ that satisfies $\bigwedge_{l \in \sigma} l$ also satisfies C, otherwise $C|_\tau = \{l \mid l \in C \text{ and } l, \neg l \notin \sigma\}$. For example, $(x_1 \vee x_2)|_{\{\neg x_1, x_2\}} = \top$, and $(x_1 \vee x_2)|_{\{\neg x_2, \neg x_3\}} = (x_1)$. Similarly the application of σ to a formula F is written $F|_\sigma$ and defined by: $F|_\sigma = \top$ if $C|_\sigma = \top$ for all $C \in F$, otherwise $F|_\sigma = \{C|_\sigma \mid C \in F \text{ and } C|_\sigma \neq \top\}$. *Unit propagation* is the iterated replacement of F with $F|_{\{l\}}$ for each unit clause $(l) \in F$, until F includes the empty clause

$\bot$, or F contains no unit clauses. A formula F implies a clause C by *reverse unit propagation* (RUP) if unit propagation on $F \wedge \neg C$ ends by producing $\bot$ [39].

For a formula F and clause C, if F and $F \wedge C$ are equisatisfiable (both satisfiable or both unsatisfiable) then C is *redundant* with respect to F. Efficiently identifiable redundant clauses are at the foundation of many formula simplification techniques and refutation systems (for instance, see [51, 54, 55, 60]). In general, deciding whether a clause is redundant is complete for the complement of the class DP [8], containing both NP and co-NP [79], so solvers and proof systems rely on polynomially-decidable *redundancy properties* for checking specific instances of redundancy. The following characterization of redundant clauses provides a common framework for formulating such properties.

Theorem 3.2.1 (Heule, Kiesl, and Biere [52])**.** *A clause $C \neq \bot$ is redundant with respect to a formula F if and only if there is a partial assignment ω such that $C|_\omega = \top$ and $F|_\alpha \vDash F|_\omega$, for the partial assignment $\alpha = \{\neg l \mid l \in C\}$.*

The partial assignment ω, usually called a *witness* for C, includes at least one of the literals occurring in C, while α is said to *block* the clause C. Redundancy properties can be defined by replacing $\vDash$ in the theorem above with efficiently-decidable relations R such that $R \subseteq \vDash$. *Propagation redundancy* (PR) [51] replaces $\vDash$ with $\vdash_1$, where $F \vdash_1 G$ if and only if F implies each $D \in G$ by RUP. The property PR gives rise to a refutation system, in which a refutation is a list of clauses $C_1, \ldots, C_n$ and witnesses $\omega_1, \ldots, \omega_n$ such that $C_k|_{\omega_k} = \top$ and $(F \bigwedge_{i=1}^{k-1} C_i)|_{\alpha_k} \vdash_1 (F \bigwedge_{i=1}^{k-1} C_i)|_{\omega_k}$ for all $1 \leq k \leq n$, and $F \bigwedge_{i=1}^{n} C_i \vdash_1 \bot$.

Most redundancy properties used in SAT solving can be understood as restricted forms of propagation redundancy. The RAT property [60] is equivalent to *literal propagation redundancy*, where the witness ω for any clause C may differ from the associated α on only one literal; that is, $\omega = (\alpha \setminus \{\neg l\}) \cup \{l\}$ for some $l \in C$ [52]. The DRAT system [99] is based on RAT, with the added ability to remove clauses from the accumulated formula $F \bigwedge C_i$.

3.3 Redundancy for Boolean Functions

Theorem 3.2.1 provides a foundation for clausal proof systems by characterizing redundant clauses in a convenient way. However, the restriction to clauses places limitations on these systems, making some forms of non-clausal reasoning difficult to express. For solvers aiming to construct refutations in these systems, this translates directly to restrictions on which solving techniques can be used.

We show this characterization can be broadened to include redundancy for non-clausal constraints, and can be used to define useful redundancy properties and refutation systems. The contributions of this paper are divided into three corresponding levels of generality. The top level, covered in the current section,

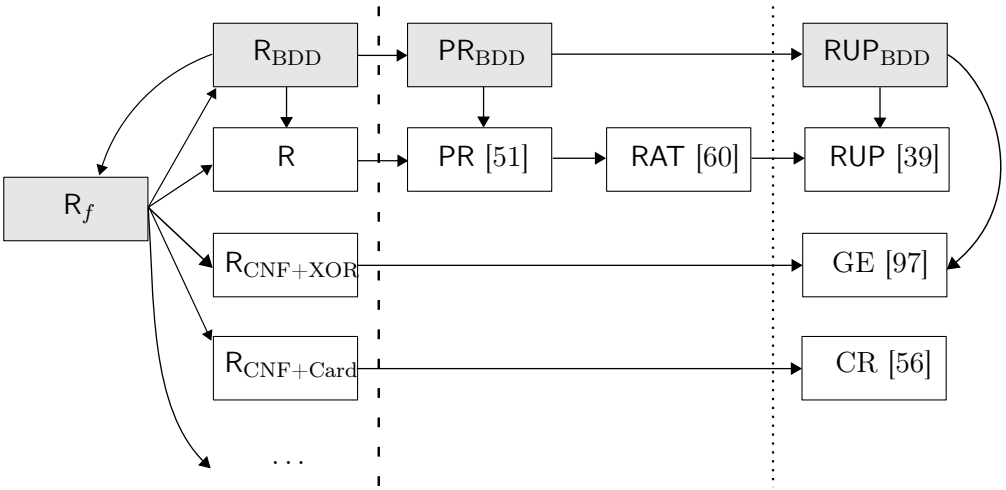


Figure 3.1: Different notions of redundancy and their relationships. An arrow from A to B indicates A generalizes B. Properties to the right of the thick dashed line are polynomially checkable; those to the right of the thin dotted line only derive logical consequences. Novel properties defined in this paper are grey.

is the direct extension of Theorem 3.2.1 from redundancy for clauses, written R, to redundancy for Boolean functions, written R_f. The middle level, the focus of Section 3.4, instantiates the resulting Theorem 3.3.1 to define the refutation systems $\mathsf{RUP}_{\mathrm{BDD}}$ and $\mathsf{PR}_{\mathrm{BDD}}$ based on redundancy for Binary Decision Diagrams. At the bottom level, these systems are shown to easily handle Gaussian elimination (GE) in Section 3.5, as well as some aspects of cardinality reasoning (CR). The relationships between these notions of redundancy are shown in Figure 3.1.

Each level of generality is individually important to this work. At the bottom level, the straightforward expression of Gaussian elimination by $\mathsf{RUP}_{\mathrm{BDD}}$ and $\mathsf{PR}_{\mathrm{BDD}}$ makes it more feasible for solvers to use this efficient technique with proof production, especially as these systems generalize their clausal analogs already in use. The results in Section 3.6 confirm the usefulness of $\mathsf{RUP}_{\mathrm{BDD}}$ for this purpose. At the middle level, we show the notion of redundancy instantiated for BDDs in this way may be capable of other strong forms of reasoning as well. Finally, the top level provides a very general form of redundancy, independent of function representation. This may make possible the design of redundancy properties and refutation systems in contexts where the BDD representation of constraints is too large; for example, it is known that some pseudo-Boolean constraints can in general have exponential size BDD representations [1, 58].

This section presents in Theorem 3.3.1 a characterization of redundancy for Boolean functions in general. One way of instantiating this characterization

is demonstrated in Section 3.4 where the functions are represented by Binary Decision Diagrams; the resulting refutation systems are shown in Section 3.5 to easily express Gaussian elimination. However, the applicability of Theorem 3.3.1 is much broader, providing a foundation for redundancy-based refutation systems independent of the representation used.

Proofs of theoretical results not included in the text can be found in the appendix.

3.3.1 Characterizing R_f

We begin with the definition corresponding to the property R_f.

Definition 3.3.1. A Boolean function g is *redundant* with respect to a Boolean function f if the functions f and $f \wedge g$ are both satisfiable, or both unsatisfiable.

As we will see, extending Theorem 3.2.1 to the non-clausal case relies on the notion of a *Boolean transformation*, or just transformation: a function $\varphi : B^V \to B^V$, mapping assignments to assignments. Importantly, for a function f and transformation φ, in fact $f \circ \varphi : B^V \to B$ is a function as well, where as usual $f \circ \varphi(\tau) = f(\varphi(\tau))$. For instance let $F = x_1 \wedge x_2$ and for all $\tau \in B^V$, the transformation φ *flips* x_1, so that $\varphi(\tau)(x_1) = \neg\tau(x_1)$, and *ignores* x_2, that is, $\varphi(\tau)(x_2) = \tau(x_2)$. Then in fact $F \circ \varphi$ is expressed by the formula $\neg x_1 \wedge x_2$.

Composing a function with a transformation can be seen as a generalization of the application of a partial assignment to a formula or clause as defined in the previous section. Specifically, for a partial assignment σ let $\hat{\sigma}$ refer to the following transformation: for any assignment τ, the assignment $\hat{\sigma}(\tau)$ satisfies $\bigwedge_{l \in \sigma} l$, and $\hat{\sigma}$ ignores any $x \in V$ such that $x, \neg x \notin \sigma$. Then for any formula F the formula $F|_\sigma$ expresses exactly the function $F \circ \hat{\sigma}$. In particular, if α is the partial assignment blocking a clause C then notice $C \circ \hat{\alpha}(\tau) = 0$ for all τ, but $\hat{\alpha}$ ignores variables not appearing in C; consequently $\hat{\alpha}(\tau) = \tau$ if τ already falsifies C. Generalizing this idea to transformations that block non-clausal constraints is more complicated. In particular, there may be multiple blocking transformations.

Example 3.3.1. Let g be the function $g(\tau) = 1$ if and only if $\tau(a) \neq \tau(b)$ (i.e. g is an XOR constraint). Transformations α_1, α_2 are shown in the table below.

$\tau(a)$	$\tau(b)$	g	$\alpha_1(\tau)(a)$	$\alpha_1(\tau)(b)$	$g \circ \alpha_1$	$\alpha_2(\tau)(a)$	$\alpha_2(\tau)(b)$	$g \circ \alpha_2$
0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	1	1	0
1	0	1	0	0	0	0	0	0
1	1	0	1	1	0	1	1	0

Both transformations ignore all $x \neq a, b$. Notice if $g(\tau) = 0$ then τ is unaffected by either transformation, and $g \circ \alpha_1(\tau) = g \circ \alpha_2(\tau) = 0$ for any assignment τ. However α_1 and α_2 are different, so that, for example, if $F = \neg a \wedge (b \vee c)$ and τ satisfies the literals $\neg a$, b, and c then $F \circ \alpha_1(\tau) = 1$ but $F \circ \alpha_2(\tau) = 0$.

Motivated by this we define transformations blocking a function as follows.

Definition 3.3.2. A transformation α *blocks* a function g if $g \circ \alpha$ is unsatisfiable, and for any assignment τ if $g(\tau) = 0$ then $\alpha(\tau) = \tau$.

Notice any g not equal to the constant function 1 has blocking transformations; for example, by mapping every τ satisfying g to a particular assignment falsifying it. Using this definition, the following theorem shows how the redundancy of a Boolean function g with respect to another function f can be demonstrated. This is a direct generalization of Theorem 3.2.1, using a transformation blocking g in the place of the partial assignment blocking a clause, and a transformation ω such that $g \circ \omega$ is the constant function 1 in place of the witnessing assignment.

Theorem 3.3.1. *Let f be a function and g a non-constant function. Then g is redundant with respect to f if and only if there exist transformations α and ω such that α blocks g and $g \circ \omega$ is the constant function 1, and further $f \circ \alpha \vDash f \circ \omega$.*

Proof. ($\Rightarrow$) Suppose g is redundant with respect to f and let α be any transformation blocking g. If f is unsatisfiable then $f \circ \alpha$ is as well, so that $f \circ \alpha \vDash f \circ \omega$ holds for any ω. Thus we can take as ω the transformation $\omega(\tau) = \tau^*$ for all $\tau \in B^V$, where τ^* is some assignment satisfying g. If instead f is satisfiable, by redundancy so is $f \wedge g$. Here we can take as ω the transformation $\omega(\tau) = \tau^*$ for all $\tau \in B^V$, where τ^* is some assignment satisfying $f \wedge g$. Then both $f \circ \omega$ and $g \circ \omega$ are the constant function 1, so that $f \circ \alpha \vDash f \circ \omega$ holds in this case as well.

($\Leftarrow$) Suppose α, ω meet the criteria stated in the theorem. We show that g is redundant by demonstrating that if f is satisfiable, then so is $f \wedge g$. Suppose τ is an assignment satisfying f. If also $g(\tau) = 1$, then of course τ satisfies $f \wedge g$. If instead $g(\tau) = 0$, then $\alpha(\tau) = \tau$ as α blocks the function g. Thus $f \circ \alpha\,(\tau) = f(\alpha(\tau)) = f(\tau) = 1$. As $f \circ \alpha \vDash f \circ \omega$, this means $f(\omega(\tau)) = 1$. As $g \circ \omega$ is the constant function 1 then $g(\omega(\tau)) = 1$, so $\omega(\tau)$ satisfies $f \wedge g$.

□

The clausal characterization in Theorem 3.2.1 shows that the redundancy of a clause can be evidenced by providing a witnessing assignment and demonstrating that an implication holds, providing a foundation for refutations based on the iterative conjunction of clauses. Theorem 3.3.1 above shows that the redundancy of a function in general can be seen in the same way by providing transformations α and ω. Consequently this suggests how to construct refutations based on the iterative conjunction of Boolean functions.

Definition 3.3.3. A sequence $\sigma = (g_1, \alpha_1, \omega_1), \ldots, (g_n, \alpha_n, \omega_n)$ is a *redundancy sequence* for a Boolean function f if:

1. α_k blocks g_k and $g_k \circ \omega_k$ is the constant function 1, for all $1 \leq k \leq n$,
2. $(f \wedge \bigwedge_{i=1}^{k-1} g_i) \circ \alpha_k \vDash (f \wedge \bigwedge_{i=1}^{k-1} g_i) \circ \omega_k$, for all $1 \leq k \leq n$.

As for clausal redundancy, refutations are intuitively based on the following: if g_1 is redundant with respect to f, and g_2 is redundant with respect to $f \wedge g_1$, then f and $f \wedge g_1 \wedge g_2$ are equisatisfiable; that is, $g_1 \wedge g_2$ is redundant with respect to f. The following holds as a direct consequence.

Proposition 3.3.1. *Let f be a Boolean function. If $(g_1, \alpha_1, \omega_1), \ldots, (g_n, \alpha_n, \omega_n)$ is a redundancy sequence for f, and $f \wedge \bigwedge_{i=1}^{n} g_i$ is unsatisfiable, then so is f.*

This shows, abstractly, how redundant Boolean functions can be used as a basis for refutations in the same way as redundant clauses.

3.3.2 Redundant BDDs

To define practical, and polynomially-checkable, refutation systems based on non-clausal redundancy in this way, we focus on a representation of Boolean functions that can be used within the framework described above. Specifically, we consider sets of BDDs in conjunction, just as formulas are sets of clauses in conjunction. Clauses are easily expressed by BDDs, and thus this representation easily expresses (CNF) formulas; this is necessary as we are typically interested in proving the unsatisfiability not of functions in general, but of (CNF) formulas. It is important to notice this is only a particular instantiation of Theorem 3.3.1, and that other representations of Boolean functions may give rise to useful and efficient systems as well.

BDDs [3, 17, 68] are compact expressions of Boolean functions in the form of rooted, directed, acyclic graphs consisting of *decision nodes*, each labeled by a variable $x \in V$ and having two children, and two *terminal nodes*, labeled by 0 and 1. The BDD for a function $f : B^V \to B$ is based on its *Shannon expansion*,

$$f \;=\; (\neg x \wedge f \circ \hat{\sigma}_0) \vee (x \wedge f \circ \hat{\sigma}_1)$$

where $\sigma_0 = \{\neg x\}$ and $\sigma_1 = \{x\}$, for $x \in V$. As is common we assume BDDs are *ordered* and *reduced*: if a node with variable label x precedes a node with label y in the graph then $x \prec y$, and the graph has no distinct, isomorphic subgraphs. Representation this way is canonical up to variable order, so that no two distinct BDDs with the same variable order represent the same Boolean function [17].

Our use of BDDs for representing non-clausal redundancy relies on the concept of *cofactors* as developed in BDD literature. The functions $f \circ \hat{\sigma}_0$ and $f \circ \hat{\sigma}_1$ are

called *literal cofactors* of f by $\neg x$ and x, respectively, and are usually written $f|_{\neg x}$ and $f|_x$. The cofactor of f by a conjunction of literals $c = l_1 \wedge \cdots \wedge l_n$ can be defined similarly, so that $f|_c = f \circ \hat{\sigma}_c$, for the partial assignment $\sigma_c = \{l_1, \ldots, l_n\}$. This notation is the same as for the application of a partial assignment to a clause or formula from Section 3.2, as the notions coincide. More precisely, if a formula F and BDD f express the same function, so do the formula $F|_{\sigma_c}$ and BDD $f|_c$.

More broadly, for BDDs f and g, a *generalized cofactor* of f by g is a BDD h such that $f \wedge g = h \wedge g$; that is, f and h agree on all assignments satisfying g. This leaves unspecified what value $h(\tau)$ should take when $g(\tau) = 0$, and various different BDD operations have been developed for constructing generalized cofactors [26, 27, 28] The *constrain* operation [27] produces for f and g, with g not equal to the always false 0 BDD, a generalized cofactor which can be seen as the composition $f \circ \pi_g$, where π_g is the transformation [91]:

$$\pi_g(\tau) = \begin{cases} \tau & \text{if } g(\tau) = 1 \\ \underset{\{\tau' \mid g(\tau')=1\}}{\arg\min} \; d(\tau, \tau') & \text{otherwise.} \end{cases}$$

The function d is defined as follows: $d(\tau, \tau') = \sum_{i=1}^{n} |\tau(x_i) - \tau'(x_i)| \cdot 2^{n-i}$, where $V = \{x_1, \ldots, x_n\}$ with $x_1 \prec \cdots \prec x_n$. Intuitively, d is a measure of distance between two assignments based on the variables on which they disagree, weighted by their position in the variable order. It is important to notice then that the transformation π_g and the resulting $f \circ \pi_g$ depend on the variable order, and may differ for distinct orders. For a conjunction of literals c, though, $f \circ \pi_c = f|_c$ regardless of the order, so that $f|_g$ refers to $f \circ \pi_g$ in general.

As the transformation π_g maps an assignment falsifying the function g to the nearest assignment (with respect to d) satisfying it, a transformation that blocks the function g can surely be obtained as follows.

Lemma 3.3.1. *If g is not equal to the constant function 1 then $\pi_{\neg g}$ blocks g.*

This form of generalized cofactor, as computed by the constrain operation, is well suited for use in redundancy-based reasoning as described above, as the transformation $\pi_{\neg g}$ depends only on g. As a consequence, for BDDs f_1 and f_2 in fact $(f_1 \wedge f_2)|_{\neg g} \equiv f_1|_{\neg g} \wedge f_2|_{\neg g}$; that is, the BDD $(f_1 \wedge f_2)|_{\neg g}$ expresses the same function as the BDD for the conjunction $f_1|_{\neg g} \wedge f_2|_{\neg g}$. Thus given a set of BDDs $f_1, \ldots, f_n$ we can represent $(f_1 \wedge \cdots \wedge f_n)|_{\neg g}$ simply by the set of cofactors $f_i|_{\neg g}$ and without constructing the BDD for the conjunction $f_1 \wedge \cdots \wedge f_n$, which is NP-hard in general. In particular, given a formula $F = C_1 \wedge \cdots \wedge C_n$ and a Boolean constraint g, the function $F|_{\neg g}$ can be represented simply by applying the constrain operation to each of the BDDs representing C_i. Therefore, from Theorem 3.3.1 we can characterize redundancy for conjunctions of BDDs, written $\mathsf{R}_{\mathrm{BDD}}$, as follows.

```
UnitProp(f_1, ..., f_n)
1    repeat
2        if f_i = 0 or f_i = ¬f_j for some 1 ≤ i, j ≤ n then
3            return "conflict"
4        if U(f_i) ≠ ∅ for some 1 ≤ i ≤ n then
5            f_j := f_j|_{⋀ U(f_i)} for all 1 ≤ j ≤ n
6    until no update to f_1, ..., f_n
```

Figure 3.2: A procedure for unit propagation over a set of BDDs

Proposition 3.3.2. *Suppose $f_1, \ldots, f_n$ are BDDs and g is a non-constant BDD. If there is a partial assignment $\{l_1, \ldots, l_k\}$ such that for $\omega = \bigwedge_{i=1}^{k} l_i$,*

$$f_1|_{\neg g} \wedge \cdots \wedge f_n|_{\neg g} \vDash f_1|_{\omega} \wedge \cdots \wedge f_n|_{\omega}$$

and $g|_{\omega} = 1$ then g is redundant with respect to $f_1 \wedge \ldots \wedge f_n$.

3.4 BDD Redundancy Properties

The previous section provided a characterization of redundancy for Boolean functions, and showed how this could be instantiated for BDDs. In this section we develop polynomially-checkable properties for showing that a BDD is redundant with respect to a conjunction of BDDs, and describe their use in refutation systems for proving the unsatisfiability of formulas.

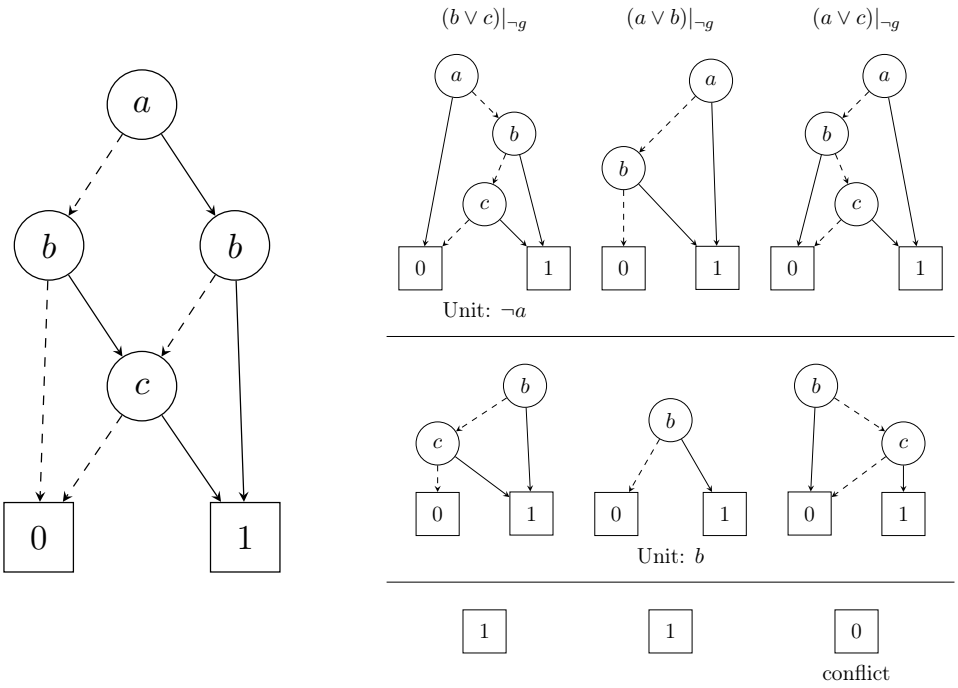
3.4.1 The UnitProp Algorithm

As Theorem 3.2.1 is used for defining clausal redundancy properties, Proposition 3.3.2 gives rise to BDD redundancy properties by replacing $\vDash$ with polynomially-decidable relations. Similar to the use of the unit propagation procedure by the clausal properties RUP and PR, we describe a unit propagation procedure for use with a set of BDDs and derive analogous properties $\mathsf{RUP}_{\mathrm{BDD}}$ and $\mathsf{PR}_{\mathrm{BDD}}$.

For a BDD f, the Shannon expansion shows that if $f|_{\neg l} = 0$ (i.e. $f|_{\neg l}$ is the always false 0 BDD) for some literal l, then $f = l \wedge f_l$, and therefore $f \vDash l$. Then the *units implied by* f, written $U(f)$, can be defined as follows.

Definition 3.4.1. $U(f) = \{l \mid var(l) \in V \text{ and } f|_{\neg l} = 0\}$, for $f : B^V \to B$.

As $f|_{\neg l}$ can be computed in $O(|f|)$, where $|f|$ is the number of nodes in the BDD for f [85], then $U(f)$ can certainly be computed in $O(|V| \cdot |f|) \subseteq O(|f|^2)$, though this can be reduced to $O(|f|)$. We write $\bigwedge U(f)$ to mean $\bigwedge_{l \in U(f)} l$.



(a) The constraint g

(b) $\mathsf{UnitProp}((b \vee c)|_{\neg g}, (a \vee b)|_{\neg g}, (a \vee c)|_{\neg g})$

Figure 3.3: Example derivation of a constraint g, shown in (a), using $\mathsf{RUP}_{\mathrm{BDD}}$. In (b), the top line shows the BDDs for each of the clauses $(b \vee c), (a \vee c), (a \vee b)$ after cofactoring by g. The second line shows each of these BDDs after cofactoring by the unit $\neg a \in U((b \vee c)|_{\neg g})$. Here, the middle BDD becomes simply the unit b, and the third line shows each BDD cofactored by the unit b. In this line, the third BDD has become 0, so a conflict is returned.

Figure 3.2 provides a sketch of the unit propagation procedure. Whenever $U(f)$ is non-empty for some f in a set of BDDs, each BDD in the set can be replaced with its cofactor by $\bigwedge U(f)$. This approach to unit propagation is largely similar to that of Olivo and Emerson [77], except we consider two conflict situations: if some BDD becomes 0, or if two BDDs are the negations of each other.

For $N = |f_1| + \cdots + |f_n|$ the procedure $\mathsf{UnitProp}(f_1, \ldots, f_n)$ can be performed in time $O(N^2)$. In line 5, if f_j and $\bigwedge U(f_i)$ share no variables, then $f_j = f_j|_{\bigwedge U(f_i)}$, otherwise the BDD for $f_j|_{\bigwedge U(f_i)}$ can be constructed in time $O(|f_j|)$ and further $|f_j|_{\bigwedge U(f_i)}| < |f_j|$. This procedure is correct: "conflict" is only returned when $\bigwedge_{i=1}^{n} f_i$ is unsatisfiable (see the appendix for the proof).

Proposition 3.4.1. *If* $\mathsf{UnitProp}(f_1, \ldots, f_n)$ *returns "conflict" then*

$$f_1 \wedge \cdots \wedge f_n \equiv 0.$$

3.4.2 The Properties $\mathsf{RUP}_{\mathrm{BDD}}$ and $\mathsf{PR}_{\mathrm{BDD}}$

UnitProp generalizes the usual unit propagation procedure on a formula: if C is a clause, then $U(C) \neq \emptyset$ implies C is a unit clause and $\bigwedge_{l \in U(C)} l = C$. We extend the relation $\vdash_1$ and the definition of RUP accordingly.

Definition 3.4.2. Let $f_1, \ldots, f_n$ and $g \neq 0$ be BDDs. Then $f_1 \wedge \cdots \wedge f_n$ implies g by $\mathsf{RUP}_{\mathrm{BDD}}$ if $\mathsf{UnitProp}(f_1|_{\neg g}, \ldots, f_n|_{\neg g})$ returns "conflict."

Example 3.4.1. Let $F = \{C_1 = b \vee c, C_2 = a \vee b, C_3 = a \vee c\}$, and assume $a \prec b \prec c$. Consider g as shown in Figure 3.3, expressing the cardinality constraint $g(\tau) = 1$ if and only if τ satisfies at least two a, b, c; also written $\{a, b, c\} \geq 2$. Figure 3.3 shows the updates made throughout $\mathsf{UnitProp}(C_1|_{\neg g}, C_2|_{\neg g}, C_3|_{\neg g})$. Notice that $U(C_1|_{\neg g}) = \{\neg a\}$, and $U((C_2|_{\neg g})|_{\neg a}) = \{b\}$. Then $C_3|_{\neg g}$ after cofactoring by $\neg a$ and b becomes the constant BDD 0, so the procedure returns "conflict." As a result, F implies the BDD g by $\mathsf{RUP}_{\mathrm{BDD}}$.

We show that $\mathsf{RUP}_{\mathrm{BDD}}$ is a redundancy property. Given BDDs $f_1, \ldots, f_n, g$, checking whether g is implied by $\mathsf{RUP}_{\mathrm{BDD}}$ primarily consists of the UnitProp procedure, though each $f_i|_{\neg g}$ must first be constructed, which can be done in time $O(|f_i| \cdot |g|)$ [27]. The size of this BDD may in some cases be larger than the size of f_i, though it is typically smaller [27, 91] and at worst $|f_i|_{\neg g}| \leq |f_i| \cdot |g|$. Consequently it can be decided in time $O(|g|^2 \cdot N^2)$ whether g is implied by $\mathsf{RUP}_{\mathrm{BDD}}$. Finally if g is implied by $\mathsf{RUP}_{\mathrm{BDD}}$ then it is redundant with respect to $f_1 \wedge \cdots \wedge f_n$; in fact, it is a logical consequence (proof available in the appendix).

Proposition 3.4.2. *If* $f_1 \wedge \cdots \wedge f_n \vdash_1 g$*, then* $f_1 \wedge \cdots \wedge f_n \vDash g$*.*

From $\mathsf{RUP}_{\mathrm{BDD}}$ the property PR can be directly generalized to this setting as well. Specifically, we define the redundancy property $\mathsf{PR}_{\mathrm{BDD}}$ as follows.

Definition 3.4.3. Suppose $f_1, \ldots, f_n$ are BDDs and g is a non-constant BDD. Then g is $\mathsf{PR}_{\mathrm{BDD}}$ with respect to $\bigwedge_{i=1}^{n} f_i$ if there is partial assignment $\{l_1, \ldots, l_k\}$ such that $g|_\omega = 1$ and $\bigwedge_{i=1}^{n} f_i|_{\neg g} \vdash_1 f_j|_\omega$ for all $1 \leq j \leq n$, where $\omega = \bigwedge_{i=1}^{k} l_i$.

Proposition 3.3.2 shows if g is $\mathsf{PR}_{\mathrm{BDD}}$ with respect to $f = f_1 \wedge \cdots \wedge f_n$ then g is redundant with respect to f, thus $\mathsf{PR}_{\mathrm{BDD}}$ is a redundancy property.

Notice these properties and derivations directly generalize their clausal equivalents; for example, if C is PR with respect to a formula F, then (the BDD expressing) C is $\mathsf{PR}_{\mathrm{BDD}}$ with respect to (the set of BDDs expressing) F. Deciding whether a clause C is PR with respect to a formula F is NP-complete [55].

As $\mathsf{PR}_{\mathrm{BDD}}$ generalizes PR, then $\mathsf{PR}_{\mathrm{BDD}}$ is NP-hard as well. Further, checking whether g is $\mathsf{PR}_{\mathrm{BDD}}$ with respect to $f_1 \wedge \cdots \wedge f_n$ by some candidate ω can be done polynomially as argued above, thus the following holds.

Proposition 3.4.3. *Deciding whether g is $\mathsf{PR}_{\mathrm{BDD}}$ with respect to $f_1 \wedge \cdots \wedge f_n$, given the BDDs $g, f_1, \ldots, f_n$, is NP-complete.*

In other words, the decision problems for PR and $\mathsf{PR}_{\mathrm{BDD}}$ are of equal complexity.

3.4.3 Refutations with BDD Redundancy Properties

The properties $\mathsf{RUP}_{\mathrm{BDD}}$ and $\mathsf{PR}_{\mathrm{BDD}}$ as defined in this section can be used to show that a BDD can be added to a set of BDDs in a satisfiability-preserving way. Of course, any clause has a straightforward and simple representation as a BDD, so that a formula can be easily represented this way as a set of BDDs. As a result $\mathsf{RUP}_{\mathrm{BDD}}$ and $\mathsf{PR}_{\mathrm{BDD}}$ can be used as systems for refuting unsatisfiable formulas. In the following, we identify a clause with its representation as a BDD, and a formula with its representation as a set of such BDDs.

To simplify the presentation of derivations based on $\mathsf{RUP}_{\mathrm{BDD}}$ and $\mathsf{PR}_{\mathrm{BDD}}$ we introduce an additional redundancy property, allowing derivations to include steps to directly derive certain BDDs *path-wise* in the following way.

Definition 3.4.4. $f_1 \wedge \cdots \wedge f_n$ implies g by $\mathsf{RUP}_{\mathrm{path}}$ if (1) $f_1 \wedge \cdots \wedge f_n \vdash_1 \neg c$ for every $c = l_1 \wedge \cdots \wedge l_m$ such that $l_1, \ldots, l_m$ is a path from the root of g to the 0 terminal, and (2) $|g| \leq \log_2(|f_1| + \cdots + |f_n|)$.

If $f_1 \wedge \cdots \wedge f_n$ implies g by $\mathsf{RUP}_{\mathrm{path}}$ then it is a logical consequence of $f_1 \wedge \cdots \wedge f_n$, as this checks that no assignment satisfies both $\neg g$ and $f_1 \wedge \cdots \wedge f_n$. The number of paths in a BDD g can however be exponential in $|g|$, as in the BDD for an XOR constraint, so the second condition ensures $\mathsf{RUP}_{\mathrm{path}}$ is polynomially-checkable.

The property $\mathsf{RUP}_{\mathrm{path}}$ is primarily useful as it allows the derivation of a BDD g whose representation as a set of clauses is included in $\{f_1, \ldots, f_n\}$: if c corresponds to a path to 0 in g, the clause $\neg c$ is included in the direct clausal translation of g. In this context, the restrictive condition (2) in Definition 3.4.4 can in fact be removed, since the number of paths in g is then at most n.

Definition 3.4.5. A sequence of BDDs $g_1, \ldots, g_n$ is a $\mathsf{RUP}_{\mathrm{BDD}}$ *derivation* from a formula F if $F \wedge \bigwedge_{i=1}^{k-1} g_i$ implies g_k by $\mathsf{RUP}_{\mathrm{BDD}}$, or by $\mathsf{RUP}_{\mathrm{path}}$, for all $1 \leq k \leq n$. A sequence of BDD and assignment pairs $(g_1, \omega_1), \ldots, (g_n, \omega_n)$ is a $\mathsf{PR}_{\mathrm{BDD}}$ *derivation* from a formula F if $F \wedge \bigwedge_{i=1}^{k-1} g_i$ implies g_k by $\mathsf{RUP}_{\mathrm{path}}$, or ω_k is a $\mathsf{PR}_{\mathrm{BDD}}$-witness for g_k with respect to $F \wedge \bigwedge_{i=1}^{k-1} g_i$, for all $1 \leq k \leq n$.

As $\mathsf{RUP}_{\mathrm{BDD}}$, $\mathsf{RUP}_{\mathrm{path}}$, and $\mathsf{PR}_{\mathrm{BDD}}$ are redundancy properties, any $\mathsf{RUP}_{\mathrm{BDD}}$ or $\mathsf{PR}_{\mathrm{BDD}}$ derivation corresponds to a redundancy sequence of the same length.

Example 3.4.2. Consider the formula $F = \{a \vee b, a \vee c, b \vee c, a \vee d, b \vee d, c \vee d\}$ and let g be the BDD such that $g(\tau) = 1$ if and only if τ satisfies at least 3 of a, b, c, d; that is, g is the cardinality constraint $\{a, b, c, d\} \geq 3$. As seen in Example 3.4.1, the constraint $g_1 = \{a, b, c\} \geq 2$ is $\mathsf{RUP}_{\mathrm{BDD}}$ with respect to F; similarly so are the constraints, $g_2 = \{a, c, d\} \geq 2$, and $g_3 = \{b, c, d\} \geq 2$. Now, $\neg a \in U(g_3|_{\neg g})$: for any τ the assignment $\pi_{\neg g}(\tau)$ satisfies at most 2 of a, b, c, d, and if a is one of them then $\pi_{\neg g}(\tau)$ surely falsifies g_3. As a result, $(g_3|_{\neg g})|_a = 0$. In a similar way $\neg b \in U(g_2|_{\neg g})$. Since $g_1|_{\neg g}$ cofactored by the units $\neg a$ and $\neg b$ is falsified, then $\mathsf{UnitProp}(g_1|_{\neg g}, g_2|_{\neg g}, g_3|_{\neg g})$ returns "conflict." Consequently g is $\mathsf{RUP}_{\mathrm{BDD}}$ with respect to $F \wedge g_1 \wedge g_2 \wedge g_3$, and g_1, g_2, g_3, g is a $\mathsf{RUP}_{\mathrm{BDD}}$ derivation from F.

This example can be generalized to show that $\mathsf{RUP}_{\mathrm{BDD}}$ is capable of expressing an inference rule for cardinality constraints called the *diagonal sum* [57]. For $L = \{l_1, \ldots, l_n\}$ let $L_i = L \setminus \{l_i\}$; the diagonal sum derives $L \geq k + 1$ from the set of all n constraints $L_i \geq k$.

While the properties and refutation systems $\mathsf{RUP}_{\mathrm{BDD}}$ and $\mathsf{PR}_{\mathrm{BDD}}$ easily extend their clausal counterparts, it is important to notice that redundancy-based systems using BDDs can be defined in other ways. For instance, say $\bigwedge_{i=1}^{n} f_i$ implies g by $\mathsf{IMP}_{\mathrm{pair}}$ if $f_i|_{\neg g} \wedge f_j|_{\neg g} = 0$ for some i, j. Then $\mathsf{IMP}_{\mathrm{pair}}$ is polynomially checkable, computing the conjunction for each pair i, j. Moreover, it is clear that $f_1 \wedge f_2 \vDash g$ if and only if $f_1 \wedge f_2$ implies g by $\mathsf{IMP}_{\mathrm{pair}}$. As many logical inference rules have this form, it is possible that systems based on $\mathsf{IMP}_{\mathrm{pair}}$ are very strong.

3.5 Gaussian Elimination

Next, we show how the Gaussian elimination technique for simplifying XOR constraints embedded in a formula is captured by the redundancy properties defined in the previous section. Specifically, if an XOR constraint X is derivable from a formula F by Gaussian elimination, we show there is a $\mathsf{RUP}_{\mathrm{BDD}}$ derivation from F including the BDD expressing X with only a linear size increase.

An *XOR clause* $[x_1, \ldots, x_n]^p$ expresses the function $f : B^V \to B$, where $V = \{x_1, \ldots, x_n\}$ and p is 0 or 1, such that $f(\tau) = 1$ if and only if the number of $x_i \in V$ satisfied by τ is equal modulo 2 to p. In other words, p expresses the parity of the positive literals x_i an assignment must satisfy in order to satisfy the XOR clause. As $[x, y, y]^p$ and $[x]^p$ express the same function, we assume no variable occurs more than once in an XOR clause. Notice that $[\,]^0$ expresses the constant function 1, while $[\,]^1$ expresses 0.

The Gaussian elimination procedure begins by detecting XOR clauses encoded in a formula F. The *direct encoding* $\mathcal{D}(X)$ of $X = [x_1, \ldots, x_n]^p$ is the collection of clauses of the form $C = \{l_1, \ldots, l_n\}$, where each l_i is either x_i or $\neg x_i$ and the number of negated literals in each C is not equal modulo 2 to p The formula

$\mathcal{D}(X)$ expresses the same function as X, containing the clauses preventing each assignment over the variables in X not satisfying X. As a result, $\mathcal{D}(X)$ implies the BDD expressing X by $\mathsf{RUP}_{\text{path}}$ (see appendix for proof).

Lemma 3.5.1. *$\mathcal{D}(X)$ implies X by $\mathsf{RUP}_{\text{path}}$, for $X = [x_1, \ldots, x_n]^p$.*

Similar to the approach of Philipp and Rebola-Pardo [80], we represent Gaussian elimination steps by deriving the addition $X \oplus Y$ of XOR clauses $X = [x_1, \ldots, x_m, z_1, \ldots, z_r]^p$ and $Y = [y_1, \ldots, y_n, z_1, \ldots, z_r]^q$, given by:

$$X \oplus Y = [x_1, \ldots, x_m, y_1, \ldots, y_n]^{p \oplus q}.$$

The following lemma shows that $X \oplus Y$ is $\mathsf{RUP}_{\text{BDD}}$ with respect to $X \wedge Y$; that is, if a $\mathsf{RUP}_{\text{BDD}}$ derivation includes X and Y then $X \oplus Y$ can be derived as well. This is a result of the following observation: while the precise cofactors of X and Y by $\neg(X \oplus Y)$ depend on the variable order $\prec$, they are the negations of one another (proof is included in the appendix).

Lemma 3.5.2. *Let v be the $\prec$-greatest variable in occurring in exactly one of X and Y, and assume v occurs in Y. Then $X|_{\neg(X \oplus Y)} = X$, and $Y|_{\neg(X \oplus Y)} = \neg X$.*

The above lemma shows that the procedure $\mathsf{UnitProp}(X|_{\neg X \oplus Y}, Y|_{\neg X \oplus Y})$ returns "conflict" immediately, and as a result $X \oplus Y$ is $\mathsf{RUP}_{\text{BDD}}$ with respect to $f_1 \wedge \cdots \wedge f_n \wedge X \wedge Y$ for any set of BDDs $f_1, \ldots, f_n$.

Define a Gaussian elimination derivation Π from a formula F as a sequence of XOR clauses $\Pi = X_1, \ldots, X_N$, such that for all $1 \leq i \leq N$, either $X_i = X_j \oplus X_k$ for $j, k < i$, or $\mathcal{D}(X_i) \subseteq F$. The size of the derivation is $|\Pi| = \sum_{i=1}^{N} s_i$, where s_i is the number of variables occurring in X_i. We show that Π corresponds to a $\mathsf{RUP}_{\text{BDD}}$ derivation with only a linear size increase. This size increase is a result of the fact that the BDD expressing an XOR clause $X = [x_1, \ldots, x_n]^p$ has size $2n + 1$ (proof of the following theorem is available in the appendix).

Theorem 3.5.1. *Suppose $\Pi = X_1, \ldots, X_N$ is a Gaussian elimination derivation from a formula F. Then there is a $\mathsf{RUP}_{\text{BDD}}$ derivation from F with size $O(|\Pi|)$.*

A consequence of this theorem is that $\mathsf{RUP}_{\text{BDD}}$ includes short refutations for formulas whose unsatisfiability can be shown by Gaussian elimination. More precisely, suppose a formula F includes the direct representations of an unsatisfiable collection of XOR clauses. Then there is a polynomial-length Gaussian elimination derivation of the unsatisfiable XOR clause $[]^1$ from F [89], and by Theorem 3.5.1, a polynomial-length $\mathsf{RUP}_{\text{BDD}}$ derivation of the unsatisfiable BDD 0.

Notably, $\mathsf{RUP}_{\text{BDD}}$ then includes short refutations of, for example, the Tseitin formulas, for which no polynomial-length refutations exist in the resolution system [92, 94]. This limitation of resolution holds as well for the clausal RUP

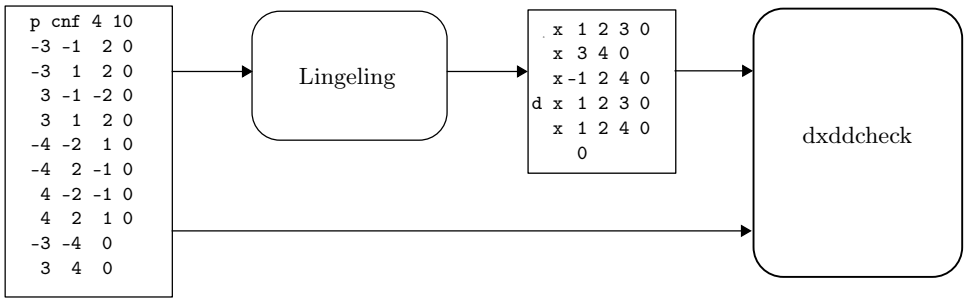


Figure 3.4: Usage of the tool dxddcheck, showing an example formula and refutation.

system, without the ability to introduce new variables, as it can be polynomially simulated by resolution [11, 36]. As the translation into $\mathsf{RUP}_{\mathrm{BDD}}$ used to prove Theorem 3.5.1 introduces no new variables, this demonstrates the strength of $\mathsf{RUP}_{\mathrm{BDD}}$ compared to resolution and its clausal analog RUP.

3.6 Results

To begin to assess the practical usefulness of the systems introduced in Section 3.4, we have implemented in Python a prototype of a tool called dxddcheck[1] for checking refutations in a subset of $\mathsf{RUP}_{\mathrm{BDD}}$. In particular we focus on the result of Section 3.5, that Gaussian elimination is succinctly captured by $\mathsf{RUP}_{\mathrm{BDD}}$.

We ran the SAT solver Lingeling (version `bcp`) on a collection of crafted unsatisfiable formulas, all of which can be solved using Gaussian elimination. From Lingeling output we extract a list of XOR clause additions and deletions, ending with the addition of the empty clause, as shown in Figure 3.4. This list is passed directly to dxddcheck, which carries it out as a $\mathsf{DRUP}_{\mathrm{BDD}}$ refutation; that is, a $\mathsf{RUP}_{\mathrm{BDD}}$ refutation also allowing steps which remove or "delete" BDDs from the set. These deletion steps can be removed without affecting the correctness of the refutation, though their inclusion can decrease the time required for checking it, as is the case with DRUP and RUP.

For these experiments we used a 1.8 GHz Intel Core i5 CPU with 8 GB of memory. The table shows the time Lingeling took to solve each formula, the number of lines in the constructed proof and its size, and the time dxddcheck took to construct and check the associated $\mathsf{DRUP}_{\mathrm{BDD}}$ proof. These benchmarks are well-known challenging examples in the contexts of XOR reasoning and proof

[1] Source code is available under the MIT license at `http://fmv.jku.at/dxddcheck` along with the benchmarks used and our experimental data.

Formula	num. of variables	num. of clauses	solving time (sec.)	num. proof lines	proof size (KB)	checking time (sec.)
rpar_50	148	394	0.1	297	7	0.34
rpar_100	298	794	0.1	597	15	1.35
rpar_200	598	1594	0.2	1197	35	6.67
mchess_19	680	2291	0.0	1077	41	4.07
mchess_21	836	2827	0.1	1317	50	5.09
mchess_23	1008	3419	0.1	1581	63	6.42
urquhart-s5-b2	107	742	0.0	150	7	0.95
urquhart-s5-b3	121	1116	0.1	150	9	1.64
urquhart-s5-b4	114	888	0.0	150	8	1.20

production. The rpar_ n formulas are compact, permuted encodings of two contradictory parity constraints on n variables, described by Chew and Heule [23]. The mchess_ n formulas are encodings of the mutilated $n \times n$-chessboard problem, as studied by Heule, Kiesl, and Biere [49] as well as Bryant and Heule [18]. The urquhart formulas [22, 93] are examples of hard Tseitin formulas.

Lingeling solved each formula by Gaussian elimination almost instantly. We ran Lingeling and Kissat [16], winner of the main track of the SAT competition in 2020, on the benchmarks without Gaussian elimination, as is required for producing clausal refutations, using an Intel Xeon E5-2620 v4 CPU at 2.10 GHz. Only rpar_50 was solved in under about 10 hours, producing significantly larger proofs; for instance, Kissat produced a refutation of size 6911 MB.

While methods to construct clausal proofs from Gaussian elimination have been proposed, most are either lacking a public implementation or are limited in scope [23, 80]. An exception is the approach very recently proposed by Gocht and Nordström using pseudo-Boolean reasoning [38], with which we are interested in carrying out a thorough comparison of results in the future.

3.7 Conclusion

We presented a characterization of redundancy for Boolean functions, generalizing the framework of clausal redundancy and efficient clausal proof systems. We showed this can be instantiated to design redundancy properties for functions given by BDDs, and polynomially-checkable refutation systems based on the conjunction of redundant BDDs, including the system $\mathsf{PR}_{\mathrm{BDD}}$ generalizing the clausal system PR. The system $\mathsf{PR}_{\mathrm{BDD}}$ also generalizes $\mathsf{RUP}_{\mathrm{BDD}}$, which can express Gaussian elimination reasoning without extension variables or clausal translations. The results of a preliminary implementation of a subset of $\mathsf{RUP}_{\mathrm{BDD}}$ confirms such refutations are compact and can be efficiently checked.

Examples 3.4.1 and 3.4.2 show $\mathsf{RUP}_{\mathrm{BDD}}$ reasoning over cardinality constraints, and we are interested in exploring rules such as *generalized resolution* [56, 57]. Other forms of non-clausal reasoning may be possible using BDD-based redundancy systems as well. We are particularly interested in exploring the property $\mathsf{IMP}_{\mathrm{pair}}$.

While the system $\mathsf{RUP}_{\mathrm{BDD}}$ derives only constraints implied by the conjunction of the formula and previously derived constraints, $\mathsf{PR}_{\mathrm{BDD}}$ is capable of *interference-based* reasoning [43], like its clausal analog PR; there are possibly novel, non-clausal reasoning techniques taking advantage of this ability. Further, $\mathsf{RUP}_{\mathrm{BDD}}$ and $\mathsf{PR}_{\mathrm{BDD}}$ are based on the conjunction of BDDs, though Theorem 3.3.1 is more general and could be used for other ways of expressing Boolean functions. Finally we are interested in developing an optimized tool for checking proofs in the system $\mathsf{PR}_{\mathrm{BDD}}$, as well as a certified proof checker.

Acknowledgements. We extend our thanks to Marijn Heule for his helpful comments on an earlier draft of this paper.

Chapter 3 Appendix

Proposition 3.3.2. *Suppose $f_1, \ldots, f_n$ are BDDs and g is a non-constant BDD. If there is a partial assignment $\{l_1, \ldots, l_k\}$ such that for $\omega = \bigwedge_{i=1}^{k} l_i$,*

$$f_1|_{\neg g} \wedge \cdots \wedge f_n|_{\neg g} \vDash f_1|_{\omega} \wedge \cdots \wedge f_n|_{\omega}$$

and $g|_{\omega} = 1$ then g is redundant with respect to $f_1 \wedge \ldots \wedge f_n$.

Proof. Let $f = f_1 \wedge \cdots \wedge f_n$ and $\alpha = \pi_{\neg g}$. We know α blocks g by Lemma 3.3.1, and $f \circ \alpha \equiv f_1|_{\neg g} \wedge \cdots \wedge f_n|_{\neg g}$. Further, let $\sigma = \{l_1, \ldots, l_k\}$ so that $f \circ \hat{\sigma}$ is equal to $f|_{\omega} \equiv f_1|_{\omega} \wedge \cdots \wedge f_n|_{\omega}$. By Theorem 3.3.1 then g is redundant with respect to f.

□

Proposition 3.4.1. *If* $\mathsf{UnitProp}(f_1, \ldots, f_n)$ *returns "conflict" then*

$$f_1 \wedge \cdots \wedge f_n \equiv 0.$$

Proof. Let f_j^k refer to the BDD f_j after k iterations of the outer loop, and let U_k refer to the conjunction of all units produced in iteration k. Then $f_j^k = f_j^{k-1}|_{U_k}$ for any $k > 0$. As $\bigwedge_{j=1}^{n} f_j^{k-1} \vDash U_k$, then $\bigwedge_{j=1}^{n} f_j^{k-1} \equiv \bigwedge_{j=1}^{n} f_j^{k-1} \wedge U_k$, also equivalent to $(\bigwedge_{j=1}^{n} f_j^{k-1})|_{U_k} \wedge U_k$. As this cofactor distributes over conjunction, this is equivalent to $\bigwedge_{j=1}^{n} f_j^{k-1}|_{U_k} \wedge U_k$. Thus we have $\bigwedge_{j=1}^{n} f_j^{k} \wedge U_k \equiv \bigwedge_{j=1}^{n} f_j^{k-1}$.

If "conflict" is returned in the first iteration, then clearly $\bigwedge_{j=1}^{n} f_j$ is unsatisfiable. By the reasoning above, after iteration k, inductively $\bigwedge_{j=1}^{n} f_j^k$ and $\bigwedge_{j=1}^{n} f_j^{k-1}$ are equisatisfiable. As a result, if "conflict" is returned after any number of iterations $k \geq 0$, then $\bigwedge_{j=1}^{n} f_j$ is unsatisfiable.

□

Proposition 3.4.2. *If $f_1 \wedge \cdots \wedge f_n \vdash_1 g$, then $f_1 \wedge \cdots \wedge f_n \vDash g$.*

Proof. If $f_1 \wedge \cdots \wedge f_n \vdash_1 g$ then $f_1|_{\neg g} \wedge \cdots \wedge f_n|_{\neg g} \equiv 0$ by Proposition 3.4.1. Then for $f = \bigwedge_{i=1}^{n} f_i$, the BDD $f|_{\neg g}$ is the constant 0. As $f|_{\neg g}$ is a generalized cofactor of f by $\neg g$, then in fact $f \wedge \neg g = f|_{\neg g} \wedge \neg g$ is unsatisfiable, and thus $f \vDash g$.

□

Lemma 3.5.1. *$\mathcal{D}(X)$ implies X by $\mathsf{RUP}_{\text{path}}$, for $X = [x_1, \ldots, x_n]^p$.*

Proof. Let $c = l_1 \wedge \cdots \wedge l_n$, with each $var(l_i) \in \{x_1, \ldots, x_n\}$, and suppose $X|_c = 0$, so that $l_1, \ldots, l_n$ is a path to 0 in X. The number of positive literals $l_i = x_i$ in c is then not equal modulo 2 to p, so the number of negative literals in the clause $\neg c$ is not equal modulo 2 to p. Then $\neg c \in \mathcal{D}(X)$ and thus $\mathcal{D}(X) \vdash_1 \neg c$.

By the Shannon expansion $X = (x_1 \wedge X|_{x_1}) \vee (\neg x_1 \wedge X|_{\neg x_1})$, where $X|_{x_1}$ and $X|_{\neg x_1}$ are the functions expressed by the XOR clauses $[x_2, \ldots, x_n]^{\neg p}$ and $[x_2, \ldots, x_n]^p$, respectively. As $[x_n]^0$ and $[x_n]^1$ are equivalent to just $\neg x_n$ and x_n, respectively, then $|X| = 2n + 1$: with $x_1 \prec \cdots \prec x_n$, the BDD X includes one node with variable x_1, and two nodes with x_i for each $1 < i \leq n$. The formula $\mathcal{D}(X)$ includes $2^n - 1$ clauses each with n literals, thus $|X| \leq \log |\mathcal{D}(X)|$.

□

Lemma 3.5.2. *Let v the $\prec$-greatest variable in occurring in exactly one of X and Y, and assume v occurs in Y. Then $X|_{\neg(X \oplus Y)} = X$, and $Y|_{\neg(X \oplus Y)} = \neg X$.*

Proof. Let $g = \neg(X \oplus Y)$ and suppose $X(\tau) = 1$. If $Y(\tau) = 0$, then $g(\tau) = 1$ and $\pi_g(\tau) = \tau$. If instead $Y(\tau) = 1$, then $g(\tau) = 0$. The nearest assignment (with respect to d) satisfying g differs only on $\tau(v)$; that is, $\pi_g(\tau)(x) = \tau(x)$ for $x \neq v$, and $\pi_g(\tau)(v) = \neg\tau(v)$. This way $Y(\pi_g(\tau)) = 0$, and as v does not occur in X then still $X(\pi_g(\tau)) = 1$. In either case, $X \circ \pi_g(\tau) = X(\tau)$ and $Y \circ \pi_g(\tau) = \neg X(\tau)$.

Next, suppose $X(\tau) = 0$. If $Y(\tau) = 1$ then $g(\tau) = 1$ and $\pi_g(\tau) = \tau$. If instead $Y(\tau) = 0$, then also $g(\tau) = 0$. Again, $\pi_g(\tau)$ need only alter the assignment of v to satisfy Y and thus g; that is, $\pi_g(\tau)(x) = \tau(x)$ for $x \neq v$, and $\pi_g(\tau)(v) = \neg\tau(v)$. Now $Y(\pi_g(\tau)) = 1$, and again $X(\pi_g(\tau)) = 0$ is unaffected. Then $X \circ \pi_g(\tau) = X(\tau)$ and $Y \circ \pi_g(\tau) = \neg X(\tau)$ holds in either case.

□

Theorem 3.5.1. *Suppose* $\Pi = X_1, \ldots, X_N$ *is a Gaussian elimination derivation from a formula* F*. Then there is a* $\mathsf{RUP}_{\mathrm{BDD}}$ *derivation from* F *with size* $O(|\Pi|)$.

Proof. By assumption $\mathcal{D}(X_1) \subseteq F$, so that by Lemma 3.5.1 F implies X_1 by $\mathsf{RUP}_{\mathrm{path}}$. Next, for $i > 1$, either $\mathcal{D}(X_i) \subseteq F$ as well, or $X_n = X_j \oplus X_k$ for $j, k < n$, and then X_n is $\mathsf{RUP}_{\mathrm{BDD}}$ with respect to $F \wedge X_1 \wedge \cdots \wedge X_{n-1}$ by Lemma 3.5.2. Thus the sequence of BDDs representing $\sigma = X_1, \ldots, X_N$ is a $\mathsf{RUP}_{\mathrm{BDD}}$ derivation from F. As $|X_i| = 2 \cdot s_i + 1$ for $1 \leq i \leq N$, then σ has size $O(|\Pi|)$.

□

Chapter 4

Redundancy by Transformation

Publication. The content of this chapter has not appeared in any prior publication. However it is written similar to the previous chapters to be self-contained.

Author. Lee A. Barnett.

Acknowledgement. Supported by the Linz Institute of Technology AI Lab funded by the State of Upper Austria, as well as the Austrian Science Fund (FWF) under project W1255-N23, the LogiCS Doctoral College on Logical Methods in Computer Science.

Abstract. Clause elimination procedures, satisfiability-preserving methods that remove clauses from formulas in conjunctive normal form, are an important collection of formula simplification techniques used by tools for solving the Boolean satisfiability (SAT) problem. Procedures that remove covered clauses, a complex but general class of redundant clauses, are strong but rely on a less efficient process for reconstructing solutions to the clauses it removes, as the partial assignments called witnesses typically used for reconstruction are hard to compute for covered clauses. This chapter shows that witnesses in the form of transformations, functions mapping assignments to other assignments, are not hard to compute for covered clauses and can be compactly described using graph structures called transformation diagrams. It explains the use of transformation diagrams for validating clause redundancy, as well as for reconstructing solutions to clauses removed by elimination procedures.

4.1 Introduction

Tools for solving the Boolean satisfiability (SAT) problem are becoming highly efficient in practice, capable of handling problems from a wide range of applications, including hardware verification [24, 40, 61], cryptanalysis [73, 74, 87], and even pure mathematics [53, 65]. Many factors play a role in the success of modern SAT solvers, including clever search algorithms and expertly engineered software, as well as the development of techniques and proof systems that take advantage of redundancy-based reasoning (for example, [50, 54, 55], see also [43]). In general, these are methods that alter the formula to be solved, such as by adding or removing clauses, while preserving whether it is satisfiable, though without necessarily preserving its set of solutions.

Clause elimination procedures, such as blocked clause elimination [59, 67], form an important set of redundancy-based methods. Employed before and throughout solving, these procedures can be crucial to solving problems derived from practical application areas [47, 54]. Covered clause elimination (CCE) [48] is a powerful technique that identifies and removes so-called covered clauses, a generalized form of blocked clauses, and has been implemented in a number of SAT solving tools [12, 13, 34, 70].

As with similar clause elimination methods, CCE can alter a formula's satisfying assignments, so that solutions found after removing clauses may not satisfy the original formula. The standard approach to repairing solutions after elimination is to record, for each removed clause C, a redundancy witness: a partial assignment, often no larger than C itself, that can replace part of the solution to ensure C is satisfied as well [51, 52]. However, CCE utilizes a more complex solution reconstruction process that can, for each C, involve multiple steps and require quadratic space [54, 60].

Witnesses exist in theory for covered clauses, as for all redundant clauses [51], but they may be large and difficult to produce. In some cases, witnesses for covered clauses must include satisfying assignments for an arbitrarily large portion of the surrounding formula; this result that was instantiated to show that even propagation redundancy (PR), a very general redundancy property at the heart of strong proof systems for SAT [51], does not capture covered clauses [8]. As deciding whether a clause is covered is simpler than solving formulas in general, this suggests that the partial assignment structure used for redundancy witnesses is not flexible enough for some classes of redundant clauses.

Recently, alternative witness structures that generalize partial assignments have been proposed. While partial assignments can be thought of as mapping variables to truth values, Buss and Thapen, for example, define substitution redundancy (SR) by using substitutions, which extend partial assignments by allowing variables to be mapped also to literals [20]. Transformations, functions

that map assignments to assignments, were used in an abstract way to discuss redundancy properties for non-clausal constraints [7].

This chapter demonstrates that transformations are not only useful in the abstract, but in fact provide a general and intuitive approach to redundancy. Graph structures called transformation diagrams, similar to Binary Decision Diagrams (BDDs) [3, 17, 68], are described and shown to be a convenient structure for representing and working with transformations. We prove that there exists a simple transformation diagram for any covered clause C that requires space linear in the size of the covering extension of C.

The remainder of this chapter is organized as follows. Section 4.2 describes necessary symbols and terminology. Section 4.3 introduces transformation diagrams and their use, while Section 4.4 demonstrates their connection to redundancy. Section 4.5 describes witnesses for covered clauses, and Section 4.6 concludes.

4.2 Preliminaries

An *assignment* is a function $\tau : V \to B$, where V is a set of variables and $B = \{0, 1\}$ is the set of Boolean truth values. A *Boolean function* is a function $f : B^V \to B$, where B^V is the set of all possible assignments from V to B. A Boolean function f is *satisfiable* if $f(\tau) = 1$ for some $\tau \in B^V$, otherwise f is *unsatisfiable*. A *transformation* is a function $\gamma : B^V \to B^V$, mapping assignments to assignments. The identity transformation is ι, so that $\iota(\tau) = \tau$ for all τ.

Literals, clauses, and formulas are defined as usual in SAT literature. The *resolvent* of clauses $l \vee C$ and $\neg l \vee D$ upon l is the clause $l \vee C \otimes_l \neg l \vee D = C \vee D$. The variable of a literal l is written $var(l)$; for instance, $var(\neg x) = x$. A literal l is negative if $l = \neg var(l)$, and positive if $l = var(l)$. Clauses can be represented by the set of their literals, and formulas by the set of their clauses. Unless otherwise stated, formulas are assumed to be in conjunctive normal form.

A *partial assignment* σ is a set of literals such that if $l \in \sigma$ then $\neg l \notin \sigma$; that is, σ is non-contradictory. Partial assignments can be composed with assignments τ: if $x \in \sigma$ then $\tau \circ \sigma(x) = 1$, while if $\neg x \in \sigma$ then $\tau \circ \sigma(x) = 0$, otherwise $x, \neg x \notin \sigma$ and $\tau \circ \sigma(x) = \tau(x)$. Similarly if ρ and σ are partial assignments then $\rho \circ \sigma = (\rho \setminus \{l \mid \neg l \in \sigma\}) \cup \sigma$. For a Boolean function f and partial assignment σ, the *application* of σ to f is written $f|_\sigma$, referring to the function $f|_\sigma(\tau) = f(\tau \circ \sigma)$. This way the application of σ to a clause C is the clause $C|_\sigma = \top$ if $\sigma \cap C \neq 0$, otherwise $C|_\sigma = \{l \in C \mid \neg l \notin \sigma\}$. Similarly the application of σ to a formula is the formula $F|_\sigma = \{C|_\sigma \mid C \in F \text{ and } C|_\sigma \neq \top\}$. When applying small σ, we may simply write the literals in σ concatenated; for example, $f|_{a\neg bc}$ instead of $f|_{\{a,\neg b,c\}}$. For a variable x, the *literal cofactors* of f on x are $f|_x$ and $f|_{\neg x}$.

A Boolean function g is *redundant* with respect to a Boolean function f if f and $f \wedge g$ are both satisfiable, or both unsatisfiable. Redundancy is mostly considered

in terms of clauses with respect to formulas. Heule, Kiesl, and Biere [51, 52] showed that a clause C is redundant with respect to a formula F if and only if there exists a partial assignment ω, called a witness, such that every assignment satisfying $F|_\alpha$ also satisfies $F|_\omega$, written $F|_\alpha \vDash F|_\omega$, where $\alpha = \{\neg l \mid l \in C\}$.

If F contains a redundant clause C then C can be removed from F while preserving satisfiability. Clause elimination procedures are formula simplification techniques that identify and remove clauses which have some property that ensure they are redundant. For example, a clause C is blocked, and thus redundant, with respect to F if there is some $l \in C$ such that, for any $\neg l \vee D \in F$, the resolvent $C \vee D$ is tautological [59, 67].

Certain procedures, such as blocked clause elimination, can result in a reduced formula which is not logically equivalent to the original formula. It is important that assignments satisfying the original formula can be reconstructed from assignments satisfying the reduced formula. Given a witness ω for each redundant clause C removed, regardless of the clause elimination procedure applied, the *reconstruction function* [32, 60] can be used to achieve an assignment satisfying F. More specifically, given a sequence σ of witness-labeled clauses $(\omega : C)$, the reconstruction function (with respect to σ) is defined as follows [32]:

$$\mathcal{R}_\epsilon(\tau) = \tau, \qquad \mathcal{R}_{\sigma\,(\omega:C)}(\tau) = \begin{cases} \mathcal{R}_\sigma(\tau) & \text{if } C|_\tau = \top \\ \mathcal{R}_\sigma(\tau \circ \omega) & \text{otherwise.} \end{cases}$$

A sequence of witness-labeled clauses σ is a *reconstruction sequence* for a set of clauses S with respect to a formula F if $\mathcal{R}_\sigma(\tau)$ satisfies $F \cup S$ for any assignment τ satisfying $F \setminus S$. If a witness if recorded for each clause C removed by clause elimination, then even combinations of various procedures can be used to simplify the formula F. Specifically, $\sigma = (\omega_1 : C_1) \cdots (\omega_n : C_n)$ is a reconstruction sequence for $\{C_1, \ldots, C_n\} \subseteq F$ if each ω_i is a witness for C_i with respect to $F \setminus \{C_1, \ldots, C_i\}$ [32].

4.3 Transformation Diagrams

Transformations have been used abstractly in characterizing non-clausal redundancy properties. In particular, it was shown that a satisfiable Boolean function g is redundant with respect to a Boolean function f if and only if there is a transformation ω such that (1) $g(\omega(\tau)) = 1$ for any τ and (2) $f \circ \alpha \vDash f \circ \omega$, where α is a transformation ensuring $\alpha(\tau)$ falsifies g, if not $g(\tau) = 0$ already [7]. However, it was not clear how $f \circ \omega$ could be computed or expressed in general, for instance if f were a formula F, or even how to express ω other than by providing a table that lists $\omega(\tau)$ for each possible assignment τ.

This section defines graph structures, similar to BDDs, that encode transformations. We show how these transformation diagrams can be used to compute the output of a transformation given an assignment, and how to compute a formula $F|_G$ equivalent to the composition of a formula F by the transformation expressed by G.

4.3.1 Definition

One way a transformation could be described is by a binary decision tree, mapping each input assignment to an output assignment listed at the corresponding leaf vertex. Decision trees can also be used to describe Boolean functions, but in both cases this representation is typically inefficient. Directed, acyclic graphs can define Boolean functions more compactly by allowing identical subgraphs in such a representation to be merged, and unnecessary vertices to be removed, as in BDDs. Transformation diagrams extend this by allowing similarities between different output assignments to be exploited as well.

A *transformation diagram* is a directed, acyclic graph G with a single root, and a single vertex of out-degree 0 called the terminal vertex. Each non-terminal vertex in G is labeled by a Boolean variable, and each edge is labeled by a truth value, 0 or 1. Each vertex has at most two outgoing edges, and if a vertex has exactly two outgoing edges, the two edges have different labels. Vertices in G with only one outgoing edge are called *output* vertices, as they correspond to changes to the input assignment present in the output assignment.

Definition 4.3.1. A *transformation diagram* is a connected, directed, and acyclic graph G with the following properties:

1. G has a single root vertex, written G.root,
2. each vertex in G has out-degree at most two,
3. each non-terminal vertex v is labeled by a Boolean variable, written v.var, and each edge by a truth value 0 or 1,
4. no vertex has two outgoing edges with the same label, and
5. if $\pi = \{(v_1, v_2), \ldots, (v_{n-1}, v_n)\}$ is a path in G and $v_i.\text{var} = v_j.\text{var}$ for some i and j, with $i < j$, then v_j is an output vertex and v_i is not an output vertex.

We use specific terms and symbols in reference to transformation diagrams. The set of variables occurring as vertex labels in G is $var(G)$. For a vertex $v \in V$, we write v.lo (respectively, v.hi) to refer to the vertex v' such that $(v, v') \in E$ and the label of (v, v') is 0 (respectively, 1). Further, for each edge in a diagram we associate a literal, the variable of which is given by the starting vertex, and is positive or negative depending on the label of the edge.

Definition 4.3.2. For an edge $(v, v') \in E$, the *edge literal* of (v, v') is the literal $L(v, v') = v.\text{var}$ if the label of (v, v') is 1, otherwise $L(v, v') = \neg(v.\text{var})$.

The definition of transformation diagrams does not enforce an order on the variable labels occurring in G. However, it ensures that a path does not have two vertices with the same variable label, unless the vertex occurring farther from the root in the path is an output vertex. This restriction simplifies the presentation of the recurrences in the remainder of this section.

4.3.2 Apply and the Transformation of a Diagram

A transformation diagram G defines a transformation γ_G in the following way.

Definition 4.3.3. Let $G = (V, E)$ be a transformation diagram and τ an assignment defined on $var(G)$. Then $\gamma_G(\tau) = \mathsf{Apply}(\tau, G.\text{root})$, where

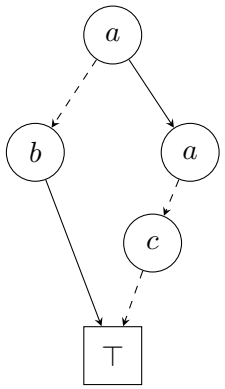
$$\mathsf{Apply}(\tau, v) = \begin{cases} \tau & \text{if } v \text{ is terminal} \\ \mathsf{Apply}(\tau \circ \{L(v, v')\}, v') & \text{if } v \text{ is an output vertex,} \\ & \text{with } (v, v') \in E \\ \mathsf{Apply}(\tau, v.\text{hi}) & \text{if } \tau(v) = 1 \\ \mathsf{Apply}(\tau, v.\text{lo}) & \text{otherwise.} \end{cases}$$

The recurrence Apply is guided through the diagram G by τ, with output vertices applying changes until reaching the terminal vertex, at which the resulting assignment is $\gamma_G(\tau)$. In other words, any assignment τ, defined on $var(G)$, corresponds to a unique path π_τ from $G.\text{root}$ to the terminal vertex. At any vertex v with two outgoing edges, the path π_τ includes the edge labeled by the value of $\tau(v.\text{var})$. Notice that if π_τ does not include an output vertex with the label x, then $\gamma_G(\tau)(x) = \tau(x)$.

A simple transformation diagram G, and table listing $\gamma_G(\tau)$ for each τ, are shown in Figure 4.1. Notice there is a transformation diagram G for any γ; for example, G could be constructed from a complete binary tree over the variables in the domain of γ, appending output vertices as needed at each leaf.

The example below describes diagrams for a common set of transformations: composing τ by a fixed partial assignment σ.

Example 4.3.1. Suppose γ is the composition of an input assignment by a given partial assignment $\sigma = \{l_1, \ldots, l_n\}$; that is, $\gamma(\tau) = \tau \circ \sigma$. We can construct a diagram $G = (V, E)$ for γ as follows. First, V consists of the terminal vertex $\top$, as well as a vertex v_i for each l_i in σ, such that $v_i.\text{var} = var(l_i)$. Then $E = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, \top)\}$, where each edge (v_i, v_j) is labeled so that $L(v_i, v_j) \in \sigma$; that is, if $v_i.\text{var} \in \sigma$ then the label of (v_i, v_j) is 1, otherwise



$\tau(a)$	$\tau(b)$	$\tau(c)$	$\gamma_G(\tau)(a)$	$\gamma_G(\tau)(b)$	$\gamma_G(\tau)(c)$
0	0	0	0	1	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	1	0
1	1	1	0	1	0

Figure 4.1: An example transformation diagram G is shown on the left. The associated transformation γ_G is shown explicitly by the table on the right. Similar to BDDs, edges labeled by 0 are indicated by dashed lines, edges labeled by 1 are solid, and the terminal vertex is labeled $\top$.

$\neg(v_i.\text{var}) \in \sigma$ and the label of (v_i, v_j) is 0. Notice that each non-terminal vertex v_i in G is an output vertex.

4.3.3 Composing Functions by Transformations with Comp

Transformations are useful in that they can be composed with Boolean functions. The remainder of this section describes how to compute, given a diagram G and a Boolean function f, the Boolean function $f \circ \gamma_G$. We provide a recurrence Comp which returns an expression equivalent to the composition of f by γ_G. Importantly, for a formula F, the expression returned by Comp is easy to convert to conjunctive normal form to get a formula equivalent to this composition, written $F|_G$.

The evaluation of Comp is similar to BDD operations for computing generalized cofactors [26, 27, 28]. At a high level, an output vertex v in a transformation diagram imposes the edge literal $L(v, v')$, for $(v, v') \in E$, as a literal cofactor on the function, while a non-output vertex $v \neq \top$ splits the function to include recursive calls on $v.\text{lo}$ and $v.\text{hi}$. These are then combined as in the dual form of the Shannon expansion of a function f; that is, $f = (\neg x \vee f|_x) \wedge (x \vee f|_{\neg x})$.

$$\mathsf{Comp}(f, v, \pi) = \begin{cases} f|_\pi & \text{if } v \text{ is terminal} \\ \mathsf{Comp}(f|_{L(v,v')}, v', \pi) & \text{if } v \text{ is an output vertex,} \\ & \text{with } (v, v') \in E \\ (x \vee \mathsf{Comp}(f, v.\text{lo}, \pi \cup \{\neg x\})) & \text{otherwise.} \\ \wedge\, (\neg x \vee \mathsf{Comp}(f, v.\text{hi}, \pi \cup \{x\})) & \end{cases}$$

An example computation of a composed function $f \circ \gamma_G$ using Comp is below.

Example 4.3.2. Let G be the transformation diagram from Figure 4.1. We refer to the vertices of G as $v_0, v_1, v_2, v_3, \top$ with $G.\text{root} = v_0$, so that $v_0.\text{var} = a$, with $v_0.\text{lo} = v_1$ and $v_0.\text{hi} = v_2$; thus $v_1.\text{var} = b$, $v_2.\text{var} = a$, and $v_3.\text{var} = c$. Then for any Boolean function f:

$$\begin{aligned}\mathsf{Comp}(f, v_0, \emptyset) &= (a \vee \mathsf{Comp}(f, v_1, \{\neg a\})) \wedge (\neg a \vee \mathsf{Comp}(f, v_2, \{a\})) \\ &= (a \vee \mathsf{Comp}(f|_b, \top, \{\neg a\})) \wedge (\neg a \vee \mathsf{Comp}(f|_{\neg a}, v_3, \{a\})) \\ &= (a \vee f|_{\neg a b}) \wedge (\neg a \vee \mathsf{Comp}(f|_{\neg a \neg c}, \top, \{a\})) \\ &= (a \vee f|_{\neg a b}) \wedge (\neg a \vee f|_{\neg a \neg c}).\end{aligned}$$

The correctness of Comp remains to be proven: specifically, that

$$f \circ \gamma_G = \mathsf{Comp}(f, G.\text{root}, \emptyset).$$

We approach this by considering the function that Comp computes at each vertex in a transformation diagram G. To be precise, for a vertex v in G let γ_v refer to the transformation $\gamma_v(\tau) = \mathsf{Apply}(\tau, v)$; that is, γ_v is the transformation corresponding to the subgraph of G that has v as a root, so that $\gamma_G = \gamma_{G.\text{root}}$. The following proposition relates Comp and γ_v, and from it the correctness of Comp follows as a direct corollary.

Proposition 4.3.1. *Let v be a vertex in a transformation diagram G, and f be a Boolean function. Further, let π be a partial assignment comprising the edge literals from non-output vertices along any path from $G.\text{root}$ to v. Then $\mathsf{Comp}(f, v, \pi)(\tau) = f(\gamma_v(\tau \circ \pi))$ for any assignment τ.*

Proof. By induction. As a base case, if v is the terminal vertex $\top$, then $\gamma_\top$ is the identity transformation ι. By definition $\mathsf{Comp}(f, v, \pi) = f|_\pi$, so that $f|_\pi(\tau) = f(\tau \circ \pi) = f(\iota(\tau \circ \pi))$ for any τ.

Now let v be a non-terminal vertex, with $v.\text{var} = x$. Suppose that v is an output vertex, so that $\gamma_v(\tau)$ is $\mathsf{Apply}(\tau, v) = \mathsf{Apply}(\tau \circ \{L(v, v')\}, v')$; in other words, $\gamma_v(\tau) = \gamma_{v'}(\tau \circ \{L(v, v')\})$. Yet x does not occur in the subgraph of G having v' as a root, which means that $\gamma_{v'}(\tau(x)) = \tau(x)$ for any τ. Consequently, $\gamma_v(\tau) = \gamma_{v'}(\tau \circ \{L(v, v')\}) = \gamma_{v'}(\tau) \circ \{L(v, v')\}$ as $var(L(v, v')) = x$. Then also $f(\gamma_v(\tau)) = f(\gamma_{v'}(\tau) \circ \{L(v, v')\}) = f|_{L(v,v')}(\gamma_{v'}(\tau))$. As v is an output vertex, $\mathsf{Comp}(f, v, \pi) = \mathsf{Comp}(f|_{L(v,v')}, v', \pi)$, and by the inductive hypothesis, $\mathsf{Comp}(f|_{L(v,v')}, v', \pi)(\tau) = (f|_{L(v,v')} \circ \gamma_{v'})(\tau \circ \pi) = f(\gamma_v(\tau \circ \pi))$.

Next suppose v is not an output vertex. First, notice that π does not contain x or $\neg x$, so $\pi \cup \{x\}$ and $\pi \cup \{\neg x\}$ are both partial assignments; that is, they do not include contradictory literals. The following hold by the inductive hypothesis:

$$\mathsf{Comp}(f, v.\mathrm{lo}, \pi \cup \{\neg x\})(\tau) = f(\gamma_{v.\mathrm{lo}}(\tau \circ (\pi \cup \{\neg x\})))$$
$$\mathsf{Comp}(f, v.\mathrm{hi}, \pi \cup \{x\})(\tau) = f(\gamma_{v.\mathrm{hi}}(\tau \circ (\pi \cup \{x\})))$$

We show that for every assignment τ, the value $f(\gamma_v(\tau \circ \pi))$ is equivalent to

$$(x \vee f(\gamma_{v.\mathrm{lo}}(\tau \circ (\pi \cup \{\neg x\})))) \wedge (\neg x \vee f(\gamma_{v.\mathrm{hi}}(\tau \circ (\pi \cup \{x\})))) \qquad (*)$$

which proves $\mathsf{Comp}(f, v, \pi) = f(\gamma_v(\tau \circ \pi))$.

Suppose $\tau(x) = 0$, so that $(*)$ is equivalent to $f(\gamma_{v.\mathrm{lo}}(\tau \circ (\pi \cup \{\neg x\})))$. As $x \notin \pi$ then $(\tau \circ (\pi \cup \{\neg x\}))(x) = 0$, so $\gamma_{v.\mathrm{lo}}(\tau \circ (\pi \cup \{\neg x\})) = \gamma_v(\tau \circ (\pi \cup \{\neg x\}))$. Further, as $x \notin \pi$ and $\tau(x) = 0$, also $\gamma_v(\tau \circ (\pi \cup \{\neg x\})) = \gamma_v(\tau \circ \pi)$. Therefore $(*)$ is equivalent to $f(\gamma_v(\tau \circ \pi))$.

Finally suppose $\tau(x) = 1$. Then $(*)$ becomes $f(\gamma_{v.\mathrm{hi}}(\tau \circ (\pi \cup \{x\})))$. As $\neg x \notin \pi$, then $\gamma_{v.\mathrm{hi}}(\tau \circ (\pi \cup \{x\})) = \gamma_v(\tau \circ (\pi \cup \{x\}))$ since $(\tau \circ (\pi \cup \{x\}))(x) = 1$. Because $\neg x \notin \pi$ and $\tau(x) = 1$ as well, then also $\gamma_v(\tau \circ (\pi \cup \{x\})) = \gamma_v(\tau \circ \pi)$. Therefore $(*)$ is equivalent to $f(\gamma_v(\tau \circ \pi))$ in this case also. □

Corollary 4.3.1. *Let G be a transformation diagram for γ_G and f a Boolean function. Then $f \circ \gamma_G = \mathsf{Comp}(f, G.root, \emptyset)$.*

Notice that if the diagram G is constructed from a partial assignment σ, as described in Example 4.3.1, then G includes only output vertices, so Comp simply applies each literal in σ as a literal cofactor to f. This means, for a formula F, the result of $\mathsf{Comp}(F, G.\mathrm{root}, \emptyset)$ is equivalent to $F|_\sigma$.

In fact, the recurrence Comp is designed so that, for a formula F, it is easy to translate the result of $\mathsf{Comp}(F, v, \pi)$ into CNF. This can be seen inductively. If v is the terminal vertex then Comp returns the formula $F|_\pi$ for a partial assignment π. If v is not terminal but is an output vertex, the computation of Comp continues with the formula $F|_{L(v,v')}$.

Now, consider the third case in the definition of Comp, and suppose both $F_0 = \mathsf{Comp}(f, v.\mathrm{lo}, \pi \cup \{\neg x\})$ and $F_1 = \mathsf{Comp}(f, v.\mathrm{hi}, \pi \cup \{x\})$ are formulas. Then the result returned by Comp can be easily translated as

$$\bigwedge_{D \in F_0} (x \vee D) \wedge \bigwedge_{D \in F_1} (\neg x \vee D).$$

For instance, recall the result in Example 4.3.2. If f is a formula F, this result is equivalent to the formula

$$\bigwedge_{D \in F|_{\neg a b}} (a \vee D) \wedge \bigwedge_{D \in F|_{\neg a \neg c}} (\neg a \vee D).$$

We write $F|_G$ to refer to this CNF translation of $\mathsf{Comp}(F, G.\text{root}, \emptyset)$. The formula $F|_G$ can be described more directly by considering the paths taken during the computation of Comp. Given a path π in G, three sets of literals can be constructed:

$$\begin{aligned} L_I(\pi) &= \{L(u,u') \mid (u,u') \in \pi \text{ and } u \text{ is not an output vertex}\} \\ L_O(\pi) &= \{L(v,v') \mid (v,v') \in \pi \text{ and } v \text{ is an output vertex}\} \\ L(\pi) &= (L_I(\pi) \setminus \{l \mid \neg l \in L_O(\pi)\}) \cup L_O(\pi). \end{aligned}$$

Each of these sets consists of edge literals occurring along π, where $L(\pi)$ in particular comprises the literals $L_O(\pi)$ from edges leaving output vertices, as well as any other edge literals along π not contradicted by a literal in $L_O(\pi)$. Then each clause in $F|_G$ has the form $\left(\bigvee_{l \in L_I(\pi)} \neg l\right) \vee D$, where $D \in F|_{L(\pi)}$, for some path π in G. As written, Comp descends each path in G until reaching the terminal vertex, so that the following holds.

Proposition 4.3.2. *If F is a formula, G a transformation diagram, and Π_G the set of paths from $G.\text{root}$ to the terminal vertex, then:*

$$F|_G = \bigwedge_{\pi \in \Pi_G} \left(\bigwedge_{D \in F|_{L(\pi)}} \left(\left(\bigvee_{l \in L_I(\pi)} \neg l \right) \vee D \right) \right) \equiv F \circ \gamma_G.$$

4.4 Diagrams As Witnesses

The previous section defined transformation diagrams, and detailed how they can be used to transform input assignments and be composed with Boolean functions. Importantly, it defined the formula $F|_G$, which can be computed using Comp from the formula F and transformation diagram G. This section focuses on the use of transformation diagrams as redundancy witnesses.

4.4.1 Transformations and Redundancy

Previously, the redundancy of a Boolean function g with respect to a Boolean function f was characterized by using two transformations α and ω [7], analogous to the partial assignments α and ω used to characterize clause redundancy by Heule, Kiesl, and Biere [51, 52]. In both cases, α is constructed to negate

the function g, or clause C, while ω must satisfy it; that is, $g \circ \omega$ is the constant function that returns the Boolean value 1, or $C|_\omega = \top$. A simplification of this result for formulas and clauses was provided by Gocht and Nordström, who require only that $F \wedge \neg C \vDash (F \wedge C)|_\omega$ [38]. Further, for the non-clausal characterization, Gocht has suggested[1] it may not be strictly necessary that the transformation α be constructed so that $g \circ \alpha$ is unsatisfiable.

Motivated by these results, we provide an intuitive and straightforward characterization of redundancy which uses only a single transformation.

Proposition 4.4.1. *Let f and g be a Boolean functions, where g is satisfiable. Then g is redundant with respect to f if and only if there is a transformation γ such that $f \vDash (f \wedge g) \circ \gamma$.*

Proof. ($\Leftarrow$) Suppose $f \vDash (f \wedge g) \circ \gamma$ for a transformation γ. If τ is an assignment satisfying f, then $\gamma(\tau)$ satisfies $f \wedge g$. If there is no such τ then f is unsatisfiable and so is $f \wedge g$. Thus g is redundant with respect to f.

($\Rightarrow$) Suppose g is redundant with respect to f; we aim to produce a transformation γ such that $f \vDash (f \wedge g) \circ \gamma$. If f is unsatisfiable then any transformation γ is suitable. If f is satisfiable, then $f \wedge g$ is as well and has a satisfying assignment τ^*. Let γ be a transformation such that $\gamma(\tau) = \tau^*$ for all τ that satisfy f. Then $f \vDash (f \wedge g) \circ \gamma$. □

The following example demonstrates this concretely. It constructs, for any redundant clause C with a partial assignment witness ω, a transformation diagram G such that γ_G acts as a witness for C in the sense of the previous proposition; that is, $F \vDash (F \wedge C)|_G$, or equivalently, $F \vDash F|_G \wedge C|_G$.

Example 4.4.1. Suppose $C = l_1 \vee \cdots \vee l_m$ is redundant with respect to F and $\omega = \{k_1, \ldots, k_n\}$ is a partial assignment such that $F|_\alpha \vDash F|_\omega$, where also $\alpha = \{\neg l_1, \ldots, \neg l_m\}$. Let $U = \{u_1, \ldots, u_m\}$ and $V = \{v_1, \ldots, v_n\}$ be sets of vertices ($U \cap V = \emptyset$) where $u_i.\text{var} = var(l_i)$ for $1 \leq i \leq m$ and $v_j.\text{var} = var(k_j)$ for $1 \leq j \leq n$. Then U contains a vertex for each literal in α, and V contains a vertex for each literal in ω. Define two sets of edges, with $\top$ denoting the terminal vertex:

$$E_\alpha = \{(u_i, u_{i+1}) \mid 1 \leq i < m\} \cup \{(u_i, \top) \mid 1 \leq i \leq m\} \cup \{(u_m, v_1)\}$$
$$E_\omega = \{(v_j, v_{j+1}) \mid 1 \leq j < n\} \cup \{(v_n, \top)\}$$

The edges in E_α are labeled such that $L(u_i, \top) = l_i$ for all $1 \leq i \leq m$, and $L(u_i, u_{i+1}) = \neg l_i$ for all $1 \leq i < m$, and further $L(u_m, v_1) = \neg l_m$. The edges in E_ω are labeled such that $L(v_j, v_{j+1}) = k_j$ for all $1 \leq j < n$ and $L(v_n, \top) = k_n$.

[1] Personal communication, March 2021.

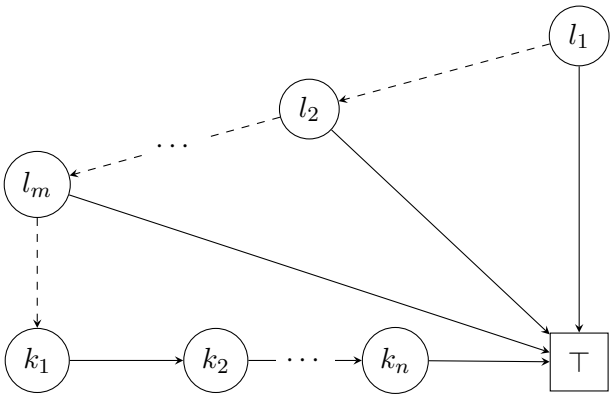


Figure 4.2: If $C = l_1 \vee \cdots \vee l_m$ is redundant with respect to a formula F, with partial assignment witness $\omega = \{k_1, \ldots, k_n\}$, then G above is a transformation diagram witness for C with respect to F; that is, $F \vDash F|_G$ and $C(\gamma_G(\tau)) = 1$ for all τ. Note that G, as shown, assumes all literals in C and ω are positive; if a literal occurs negatively, the labels on its outgoing edge(s) should be swapped.

Let $G = (U \cup V \cup \{\top\}, E_\alpha \cup E_\omega)$, with $G.\text{root} = u_1$. Figure 4.2 provides an illustration of G. We show that G defines a transformation which is a witness for C with respect to F.

First, let τ be an assignment assumed to be defined on $var(G)$. If τ satisfies C then the path π_τ in G corresponding to τ includes no output vertices, so $\gamma_G(\tau) = \tau$ satisfies C. On the other hand, if τ falsifies C then $\pi_\tau = \pi^*$, where

$$\pi^* = \{(u_1, u_2), (u_2, u_3), \ldots, (u_{m-1}, u_m), (u_m, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n), (v_n, \top).$$

Then $\gamma_G(\tau)$ satisfies each literal in ω; as ω is witness for C then $\gamma_G(\tau)$ satisfies C as well. Therefore $C(\gamma_G(\tau)) = 1$ for all τ; that is, $C|_G = \top$.

Next, we show that $F \vDash F|_G$. For $p \leq m$, define the partial assignment $\alpha_p = \{\neg l_i \mid 1 \leq i \leq p-1\} \cup \{l_p\}$. Then the formula $F|_G$ is

$$\bigwedge_{1 \leq p \leq m} \left(\bigwedge_{D \in F|_{\alpha_p}} \left(\left(\bigvee_{1 \leq i \leq p-1} l_i \right) \vee \neg l_p \vee D \right) \right) \wedge \bigwedge_{D \in F|_\omega} \left(\left(\bigvee_{1 \leq j \leq k} l_j \right) \vee D \right).$$

More simply, each clause in $F|_G$ has the form $L \vee D$, where $D \in F|_{\alpha_p}$ for some p, or $D \in F|_\omega$. We show that $F \vDash L \vee D$ for each clause $L \vee D$ in $F|_G$. First suppose $D \in F|_{\alpha_p}$, which means that $L = l_1 \vee \cdots \vee l_{p-1} \vee \neg l_p$. Then $F \wedge \neg L \vDash F|_{\alpha_p}$, so $F \wedge \neg L \vDash D$ and thus $F \vDash L \vee D$. On the other hand, suppose $D \in F|_\omega$, which means $L = C$. Then $F \wedge \neg L \vDash F|_\alpha$, and by assumption $F|_\alpha \vDash F|_\omega$, so $F|_\alpha \vDash D$. Thus $F \wedge \neg L \vDash D$, so $F \vDash L \vee D$.

We extend the term "witness" to also refer to transformation diagrams, so that a transformation diagram G such that $F \vDash (F \wedge C)|_G$ is a witness for the clause C with respect to F.

4.4.2 Checking $F \vDash F|_G$

The previous example also illustrates a useful observation about validating that the implication $F \vDash F|_G$ holds between the original formula F and the composed formula $F|_G$, for a transformation diagram G: paths in G containing no output vertices do not need to be checked. More precisely, we need only show F entails the clauses $\left(\bigvee_{l \in L_I(\pi)} \neg l\right) \vee D$ in $F|_G$ for paths π which include output literals, so that $L_I(\pi) \neq L(\pi)$.

The following proposition proves that this observation holds in general.

Proposition 4.4.2. *Let F be a formula, G a transformation diagram. Further let Π^*_G be the set of paths π in G such that $L_I(\pi) \neq L(\pi)$. Then $F \vDash F|_G$ if and only if*

$$F \vDash \bigwedge_{\pi \in \Pi^*_G} \left(\bigwedge_{D \in F|_{L(\pi)}} \left(\left(\bigvee_{l \in L_I(\pi)} \neg l \right) \vee D \right) \right).$$

Proof. ($\Rightarrow$) Let Π_G be the set of all paths in G as before, so that $\Pi^*_G \subseteq \Pi_G$. Certainly if $F \vDash F|_G$ then the implication above holds.

($\Leftarrow$) Suppose that $\pi \in \Pi_G \setminus \Pi^*_G$, and therefore $L_I(\pi) = L(\pi)$. Notice that $F \wedge \bigwedge_{l \in L_I(\pi)} l \vDash F|_{L_I(\pi)}$, which means $F \vDash \left(\bigvee_{l \in L_I(\pi)} \neg l\right) \vee D$ for all $D \in F|_{L(\pi)}$. Thus if the implication above holds then also $F \vDash F|_G$. □

We illustrate this further by providing a concrete transformation diagram, and showing that it is a witness for a redundant clause C.

Example 4.4.2. Let $C = (a \vee b)$, and

$$F = (\neg a \vee \neg b \vee c) \wedge (\neg a \vee x) \wedge (\neg c \vee \neg x \vee y) \wedge (\neg x \vee y) \wedge (\neg b \vee \neg y).$$

We show that C is redundant with respect to F by providing a transformation diagram G, shown in Figure 4.3, such that $F \vDash (F \wedge C)|_G$. First, notice that for any τ, the assignment $\gamma_G(\tau)$ satisfies C: if τ already satisfies a or b then $\gamma_G(\tau) = \tau$, otherwise $\gamma_G(\tau)$ satisfies a. Consequently $C \circ \gamma_G = C|_G = \top$. Next, to show $F \vDash F|_G$ notice there are only two paths in $\pi \in \Pi^*_G$ such that $L_I(\pi) \neq L(\pi)$. Thus $F \vDash F|_G$ if

$$F \vDash \bigwedge_{D \in F|_{a\neg bx}} (a \vee b \vee \neg x \vee D) \bigwedge_{D \in F|_{a\neg bxy}} (a \vee b \vee x \vee D).$$

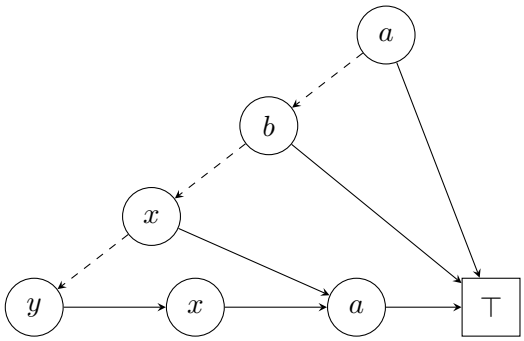


Figure 4.3: A transformation diagram which is a witness for the clause in Example 4.4.2

Simplifying these conjunctions on the right-hand side gives:

$$\bigwedge_{D \in F|_{a\neg bx}} (a \vee b \vee \neg x \vee D) = (a \vee b \vee \neg x \vee \neg c \vee y) \wedge (a \vee b \vee \neg x \vee y)$$

$$\bigwedge_{D \in F|_{a\neg bxy}} (a \vee b \vee x \vee D) = \top$$

Each of the clauses above are subsumed by clauses in F, so the implication holds. Therefore $F \vDash (F \wedge C)|_G$, which means C is redundant with respect to F.

4.4.3 Solution Reconstruction

We consider the use of transformation diagram witnesses for solution reconstruction. First, transformation diagrams can be used in the standard reconstruction function alongside partial assignment witnesses. This can be done by generalizing witness-labeled clauses to allow either transformation diagram witnesses, or partial assignment witnesses as usual.

More specifically, we can define the *combined reconstruction function* $\mathcal{R}^{\text{comb}}$ with respect to a sequence σ as follows, and show that it can be used for solution reconstruction.

$$\mathcal{R}^{\text{comb}}_{\epsilon}(\tau) = \tau, \qquad \mathcal{R}^{\text{comb}}_{\sigma\,(\omega:C)}(\tau) = \begin{cases} \mathcal{R}^{\text{comb}}_{\sigma}(\tau) & \text{if } C|_\tau = \top \\ \mathcal{R}^{\text{comb}}_{\sigma}(\tau \circ \omega) & \text{if } \omega \text{ is a partial assignment} \\ \mathcal{R}^{\text{comb}}_{\sigma}(\mathsf{Apply}(\tau, \omega.\text{root})) & \text{otherwise.} \end{cases}$$

Proposition 4.4.3. *Let F be a formula, and τ an assignment satisfying the formula $F \setminus \{C_1, \ldots, C_n\}$. Let also $\sigma = (\omega_1 : C_1) \cdots (\omega_n : C_n)$ be a sequence such*

that, for each i, either ω_i is a partial assignment witness, or ω_i is a transformation diagram witness, for the clause C_i with respect to $F \setminus \{C_1, \ldots, C_i\}$. Then $\mathcal{R}^{\text{comb}}_{\sigma}(\tau)$ satisfies F.

Proof. By induction on n, the length of σ. As a base case, if $n = 0$ then $\sigma = \epsilon$ is empty and $\mathcal{R}^{\text{comb}}_{\epsilon}(\tau) = \tau$ satisfies $F \setminus \emptyset = F$ by assumption.

For the inductive step, let $\sigma = \sigma' \cdot (\omega_n : C_n)$, so that σ' has length $n - 1$. Let also $F' = F \setminus \{C_1, \ldots, C_n\}$. There are two cases:

1. τ satisfies C_n, and thus τ satisfies $F' \wedge C_n = F \setminus \{C_1, \ldots, C_{n-1}\}$. Then by the inductive hypothesis $\mathcal{R}^{\text{comb}}_{\sigma}(\tau) = \mathcal{R}^{\text{comb}}_{\sigma'}(\tau)$ satisfies F.

2. τ falsifies C_n. As stated ω_n is either a partial assignment witness for C_n or a transformation diagram witness for C_n. If ω_n is a partial assignment, then $\tau \circ \omega_n$ satisfies $F' \wedge C_n = F \setminus \{C_1, \ldots, C_{n-1}\}$. By the inductive hypothesis $\mathcal{R}^{\text{comb}}_{\sigma}(\tau) = \mathcal{R}^{\text{comb}}_{\sigma'}(\tau \circ \omega_n)$ satisfies F. Otherwise ω_n is a transformation diagram, so that $F' \vDash (F' \wedge C_n)|_{\omega_n}$. This means the assignment $\mathsf{Apply}(\tau, \omega_n.\text{root})$ satisfies $F' \wedge C_n = F \setminus \{C_1, \ldots, C_{n-1}\}$. In this case as well, the inductive hypothesis ensures $\mathcal{R}^{\text{comb}}_{\sigma}(\tau) = \mathcal{R}^{\text{comb}}_{\sigma'}(\mathsf{Apply}(\tau, \omega_n.\text{root}))$ satisfies F.

□

However, the use of transformation diagrams allows a more straightforward definition of the reconstruction function. If ω is a partial assignment witness for a clause C, then $\tau \circ \omega$ is only guaranteed to satisfy C if τ falsifies C. On the other hand, if ω is a transformation diagram witness for C with respect to F, then $\gamma_\omega(\tau)$ satisfies $F \wedge C$ for any τ satisfying F. Intuitively, the case split in the body of the reconstruction function can be handled by Apply on ω.

In this sense, let σ be a sequence of witness-labeled clauses $(\omega : C)$ such that ω is a transformation diagram witness for C. We define the function $\mathcal{R}^{\text{TD}}$, with respect to σ, as follows:

$$\mathcal{R}^{\text{TD}}_{\epsilon}(\tau) = \tau, \qquad \mathcal{R}^{\text{TD}}_{\sigma\,(\omega:D)}(\tau) = \mathsf{Apply}(\tau, \omega.\text{root}).$$

The following proposition shows that $\mathcal{R}^{\text{TD}}$ can be used to reconstruct solutions after removing redundant clauses.

Proposition 4.4.4. *Let F be a formula, and τ an assignment satisfying the formula $F \setminus \{C_1, \ldots, C_n\}$. Let also $\sigma = (\omega_1 : C_1) \cdots (\omega_n : C_n)$ be a sequence such that each ω_i is a transformation diagram witness for the clause C_i with respect to $F \setminus \{C_1, \ldots, C_i\}$. Then $\mathcal{R}^{\text{TD}}_{\sigma}(\tau)$ satisfies F.*

Proof. By induction on n, the length of σ. As a base case, if $n = 0$ then $\sigma = \epsilon$ is empty and $\mathcal{R}^{\text{TD}}_{\epsilon}(\tau) = \tau$ satisfies $F \setminus \emptyset = F$ by assumption.

For the inductive step, let $\sigma = \sigma' \cdot (\omega_n : C_n)$, so that σ' has length $n-1$. Let also $F' = F \setminus \{C_1, \ldots, C_n\}$. Then $F' \vDash (F' \wedge C_n)|_{\omega_n}$, as ω_n is a transformation diagram witness for C_n with respect to F'. This means the assignment $\mathsf{Apply}(\tau, \omega_n.\text{root})$ satisfies $F' \wedge C_n = F \setminus \{C_1, \ldots, C_{n-1}\}$. By the inductive hypothesis, then $\mathcal{R}^{\text{TD}}_{\sigma}(\tau) = \mathcal{R}^{\text{TD}}_{\sigma'}(\mathsf{Apply}(\tau, \omega_n.\text{root}))$ satisfies F. □

4.5 Witnesses for Covered Clauses

This section describes transformation diagrams which are witnesses for covered clauses. We begin by reviewing covered literals and the covered clause property, as introduced by Heule, Järvisalo, and Biere [48].

4.5.1 Covered Clauses Summary

The set of *resolution candidates* of C in F upon a literal l, written $\text{RC}(F, C, l)$, is defined as the collection of clauses in F with which C has a non-tautological resolvent upon l.

$$\text{RC}(F, C, l) = \{C' \vee \neg l \in F \mid C' \vee \neg l \otimes_l C \not\equiv \top\}.$$

The *resolution intersection* of C in F upon l, written $\text{RI}(F, C, l)$, are the literals occurring in all of the resolution candidates, apart from $\neg l$:

$$\text{RI}(F, C, l) = \left(\bigcap \text{RC}(F, C, l)\right) \setminus \{\neg l\}.$$

Any literals in $\text{RI}(F, C, l)$ are said to be covered by l, and can be added to C while preserving satisfiability: the partial assignment $\omega = \{\neg k \mid k \in C \text{ and } k \neq l\} \cup \{l\}$ is a witness for C with respect to $F \wedge (C \cup \text{RI}(F, C, l))$. A clause C is covered in a formula F if iteratively extending C by adding covered literals results in a clause C_{EXT} that is blocked in F. If C is covered in F, then C is redundant with respect to F [48].

A generalization of covered clauses also allows the extension of C by adding asymmetric literals, where a literal k is asymmetric to C in F if there exists a clause $C' \vee \neg k \in F$ with $C' \subseteq C$. Adding asymmetric literals to C is in fact model-preserving, so that F and $(F \setminus \{C\}) \wedge (C \vee k)$ are equivalent, for any k asymmetric to C [47]. Asymmetric literals added to a clause C will not themselves cover new literals, but can allow other literals in C to cover new literals or to block C. A clause C is asymmetrically covered in F if the iterative extension of C by the addition of both covered literals and asymmetric literals is blocked in F, or subsumed; that is, the extension $C_{\text{EXT}} \subseteq C'$ for some $C' \in F$. If C is asymmetrically covered in F, then C is redundant with respect to F [48].

Definition 4.5.1. A clause C' is an *ACC extension* of C with respect to F if C' can be constructed from C by the iterative addition of covered and asymmetric literals. If some ACC extension of C is blocked or subsumed in F, then C is an *asymmetric covered clause* (ACC).

Unlike similar procedures, such as blocked clause elimination [59, 67], the procedure for identifying asymmetrically covered clauses does not provide witnesses for these clauses. Instead, it produces a sequence σ of witness-labeled clauses. The final element in σ provides a witness for the clause C_{EXT}, an ACC extension of C, with respect to F. For any other $(\omega : D) \in \sigma$ the partial assignment ω is a witness for D with respect to the formula $F \wedge (D \cup \text{RI}(F, D, l))$ for some $l \in D$. Thus, using the reconstruction function, the sequence σ can be used to reconstruct solutions to $F \wedge C$ from solutions to F [8, 48]. This sequence σ can require space quadratic in the length of the extended clause [8].

However, any clause C that is asymmetrically covered in a formula F is redundant with respect to F, so a single partial assignment ω satisfying C such that $F|_\alpha \vDash F|_\omega$ surely exists, where $\alpha = \{\neg l \mid l \in C\}$. Specifying such a partial assignment ω poses a challenge for covered clauses; specifically, in some cases ω must satisfy other clauses in F that in fact can be chosen arbitrarily. This is demonstrated by the following example, where producing a partial assignment witness ω for C requires finding a satisfying assignment to a possibly complex formula G [8].

Example 4.5.1 (Barnett, Cerna, and Biere [8])**.** The clause $C = k \vee l$ is covered in the formula:

$$\begin{array}{rcl} F \;=\; (x \vee \neg k) \wedge (\neg x \;\vee \neg y) \wedge (y \vee \neg l) & \wedge \\ (x \vee D_1) \wedge \quad \cdots \quad \wedge (x \vee D_n) & \wedge \\ (y \vee D'_1) \wedge \quad \cdots \quad \wedge (y \vee D'_n). & \end{array}$$

For each i, the clauses D_i and D'_i are variable-renamed copies of one-another, and do not include any of the variables k, l, x, or y. If $G = D_1 \wedge \cdots \wedge D_n$ is unsatisfiable, then so are F and $F|_\alpha$, which means any ω is a witness for C. On the other hand, if G is satisfiable then in fact ω must satisfy either the formula G itself, or the formula $G' = D'_1 \wedge \cdots \wedge D'_n$. Either way, any witness ω for C with respect to F indicates a how a satisfying assignment can be constructed for the formula G.

4.5.2 Diagrams for Covered Clauses

While partial assignment witnesses present this difficulty for covered clauses, there are simple and intuitive transformation diagrams that are witnesses for covered clauses. For instance, in Example 4.4.2 the clause C is covered with

respect to the formula F, and a transformation diagram witness for it is provided and shown to be correct.

In the remainder of this section, we demonstrate that for any clause C asymmetrically covered in a formula F, there is a transformation diagram G such that G is a witness for C with respect to F. The following definition provides a detailed description of the construction of this diagram G. An example of this construction is supplied after this definition.

Definition 4.5.2. For $C = l_1 \vee \cdots \vee l_m$ and formula F, let C_{EXT} be an extension of C by adding covered literals and asymmetric literals. Further, let $l_{m+1}, \ldots, l_n \in C_{\text{EXT}} \setminus C$ be the literals in C_{EXT} which were added as covered literals, ordered so that if l_j was added as a covered literal of l_i, then $i < j$. If C_{EXT} is blocked in F, then b indicates a literal l_b blocking C_{EXT}. If C_{EXT} is instead subsumed in F, then $b = n$. We define the transformation diagram $G(C, C_{\text{EXT}}, b)$ as follows. First, let V be the set of vertices

$$V = \{v_i \mid 1 \leq i \leq n\} \cup \{u_i \mid 1 \leq i \leq n\}.$$

Each vertex in V is labeled by the variable of the corresponding literal; that is, for each literal $l_i \in C_{\text{EXT}}$ there are distinct vertices u_i and v_i such that their labels are $u_i.\text{var} = v_i.\text{var} = \mathit{var}(l_i)$. The root vertex is $G(C, C_{\text{EXT}}, b).\text{root} = v_1$ corresponding to a literal $l_1 \in C$.

Next, let $E = E^- \cup E^+$ be a collection of edges, composed of two sets. First,

$$\begin{aligned} E^+ = \{(v_i, \top) \mid 1 \leq i \leq m\} \cup \{(u_i, \top) \mid 1 \leq i \leq m\} \cup \\ \{(v_i, u_j) \mid l_j \text{ covers } l_i\} \cup \{(u_i, u_j) \mid l_j \text{ covers } l_i\} \end{aligned}$$

Intuitively, E^+ includes edges corresponding to the covering relationships between the literals in C_{EXT}, as well as edges from both v_i and u_i to the terminal vertex $\top$, for each literal l_i in the clause C.

The label of each edge in E^+ matches the corresponding variable's polarity in C_{EXT}. More precisely, each $(v_i, \top) \in E^+$ is labeled so that $L(v_i, \top) \in C_{\text{EXT}}$, and likewise $L(u_i, \top) \in C_{\text{EXT}}$ for each $(u_i, \top) \in E^+$. Similarly, both $L(v_i, u_j) \in C_{\text{EXT}}$ for $(v_i, u_j) \in E^+$ and $L(u_i, u_j) \in C_{\text{EXT}}$ for $(u_i, u_j) \in E^+$.

Next, the set E^- is defined as

$$E^- = \begin{cases} \{(v_{i-1}, v_i) \mid 2 \leq i \leq n\} & \text{if } b = n \\ \{(v_{i-1}, v_i) \mid 2 \leq i \leq n\} \cup \{(v_n, u_b)\} & \text{otherwise.} \end{cases}$$

Intuitively, E^- includes a sequence of edges along the literals v_i, for each literal l_i in C_{EXT}, in order.

The label of each edge in E^- matches the negation of the corresponding variable's polarity in C_{EXT}. More precisely, $\neg L(v, v') \in C_{\text{EXT}}$ for all $(v, v') \in E^-$.

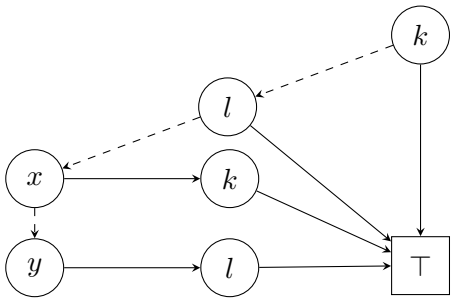


Figure 4.4: A witness for the covered clause $k \vee l$ with respect to the formula F in Example 4.5.1.

Finally, let $V' \subseteq V$ be the result of removing from V any vertices with no incoming edges, except for the root $v_1 \in V'$. Likewise,

$$E' = \{e \in E \mid \text{both vertices in } e \text{ belong to } V'\}.$$

The transformation diagram is $G(C, C_{\text{EXT}}, b) = (V', E')$.

The following example walks through the construction of the transformation diagram $G(C, C_{\text{EXT}}, b)$ corresponding to the clause C and formula provided in Example 4.5.1, show that it is a witness for C. An illustration of the resulting graph is provided in Figure 4.4.

Example 4.5.2. Let C and F be given as in Example 4.5.1, so that $C = k \vee l$ and $C_{\text{EXT}} = k \vee l \vee x \vee y$. Let $l_1 = k$, $l_2 = l$, $l_3 = x$, and $l_4 = y$. Both literals $l_3 = x$ and $l_4 = y$ are blocking literals for C_{EXT}; we choose to take $b = 4$. Following Definition 4.5.2, the set V is

$$V = \{v_1, v_2, v_3, v_4, u_1, u_2, u_3, u_4\}.$$

The literal $l_1 = k$ covers $l_3 = x$ and $l_2 = l$ covers $l_4 = y$, thus:

$$E^+ = \{(v_1, \top), (v_2, \top), (u_1, \top), (u_2, \top), (v_3, u_1), (v_4, u_2), (u_3, u_1), (u_4, u_2)\}$$
$$E^- = \{(v_1, v_2), (v_2, v_3), (v_3, v_4)\}.$$

The root vertex is v_1 but u_3 and u_4 have no incoming edges, so these are removed to result in $V' = \{v_1, v_2, v_3, v_4, u_1, u_2\}$, as well as

$$E' = \{(v_1, \top), (v_2, \top), (u_1, \top), (u_2, \top), (v_3, u_1), (v_4, u_2), (v_1, v_2), (v_2, v_3), (v_3, v_4)\}.$$

To see that $G = (V', E')$ is a witness for C with respect to F, notice that for every τ, the assignment $\gamma_G(\tau)$ satisfies either k or l. Therefore $C \circ \gamma_G = C|_G = \top$.

Next, there are only two paths to the terminal vertex in G that include output vertices, so to show $F \vDash F|_G$ we must show that

$$F \vDash \bigwedge_{D \in F|_{k \neg l x}} (k \vee l \vee \neg x \vee D) \bigwedge_{D \in F|_{\neg k l \neg x y}} (k \vee l \vee x \vee D)\,.$$

Simplifying these conjunctions gives:

$$\bigwedge_{D \in F|_{k \neg l x}} (k \vee l \vee \neg x \vee D) = (k \vee l \vee \neg x \vee \neg y) \wedge (k \vee l \vee \neg x \vee y \vee D'_1) \wedge \cdots \wedge (k \vee l \vee \neg x \vee y \vee D'_n)$$

$$\bigwedge_{D \in F|_{\neg k l \neg x y}} (k \vee l \vee x \vee D) = (k \vee l \vee x \vee D_1) \wedge \cdots \wedge (k \vee l \vee x \vee D_n)$$

Each of these clauses are subsumed by clauses in F, so the implication holds. Therefore $F \vDash (F \wedge C)|_G$, which means G is a witness for C with respect to F.

We prove now that the graph construction described in Definition 4.5.2 defines a transformation diagram witness for any asymmetric covered clause.

Lemma 4.5.1. *Let C be a clause, F a formula, and C_{EXT} be an ACC extension of C in F. If C_{EXT} is blocked or subsumed in a formula F, then $F \vDash C|_G$, where $G = G(C, C_{EXT}, b)$.*

Proof. We show that if τ satisfies F, then τ satisfies $C(\gamma_G(\tau))$. If τ satisfies a literal $l_i \in C$, then π_τ includes the edge $(v_i, \top)$. This means π_τ includes no edges from output vertices, so that $\gamma_G(\tau) = \tau$ satisfies l_i and thus C as well.

Suppose τ falsifies C but satisfies C_{EXT}. Although $\{l_1, \ldots, l_n\}$ does not include every literal in C_{EXT}, it is not possible for τ to satisfy some $l \in C_{\text{EXT}}$ but falsify each l_i. This is because each literal l in $C_{\text{EXT}} \setminus \{l_1, \ldots, l_n\}$ is asymmetric to C_{EXT}: that is, F includes some clause $C' \vee \neg l$ for $C' \subseteq C_{\text{EXT}}$. Thus if τ satisfies C_{EXT}, then it satisfies l_i for some $m < i \leq n$.

Let i be the least value such that τ satisfies l_i, so that the path π_τ begins with the edges $(v_1, v_2), \ldots, (v_{i-1}, v_i)$. As l_i was added as a covered literal of some l_j, then π_τ continues with a sequence of edges from output vertices, starting with (v_i, u_j) until $(u_k, \top)$ for some $k \leq m$. Moreover, the edges along this sequence of output vertices belong to E^+, which means $L(u_k, \top) = l_k$. Therefore $\gamma_G(\tau)$ satisfies l_k and thus C.

Finally, suppose τ falsifies C_{EXT}. Then the path π_τ begins by including the edges $(v_1, v_2), \ldots, (v_{n-1}, v_n)$. From there, as in the previous case, π_τ includes a sequence of edges from output vertices, ending with $(u_k, \top)$ for some $k \leq m$. Again, the edges along this sequence of output vertices belong to E^+, so that $\gamma_G(\tau)$ satisfies the literal $L(u_k, \top) = l_k$. □

Lemma 4.5.2. *Let C be a clause, F a formula, and C_{EXT} be an ACC extension of C in F. If C_{EXT} is blocked or subsumed in a formula F, then $F \models F|_G$, where $G = G(C, C_{EXT}, b)$.*

Proof. As in the description of G we let $C = l_1 \vee \cdots \vee l_m$ with the set of literals $\{l_{m+1}, \ldots, l_n\}$ added as covered literals in C_{EXT}. Also if C_{EXT} is blocked in F then b indicates a blocking literal l_b, otherwise $b = n$.

We show F implies each clause $\bigvee_{l \in L_I(\pi)} \neg l \vee D$ in $F|_G$, where $D \in F|_{L(\pi)}$, and π is a path in G such that $L(\pi) \neq L_I(\pi)$. For G in particular, this means the path π includes at least all the edges such that $\{\neg l \mid l \in C\} \subseteq L_I(\pi)$; in other words, beginning at the root vertex v_1, the path π must include the edge (v_m, v_{m+1}) before eventually reaching the terminal $\top$.

There are two cases, depending on the path π takes from v_{m+1}.

1. There is some k_0, with $m < k_0 \leq n$, such that

$$L_I(\pi) = \{\neg l_i \mid 1 \leq i < k_0\} \cup \{l_{k_0}\}.$$

   In other words, the path π includes the edge (v_{k_0}, u_{k_1}), for a literal l_{k_1} in C_{EXT} that covers l_{k_0}. From the vertex u_{k_1} the path π includes only edges from output vertices, so that $L_O(\pi) = \{l_{k_1}, \ldots, l_{k_p}\}$, where each l_{k_j} covers $l_{k_{j-1}}$, for $1 \leq j \leq p$.

   Now let $D \in F|_{L(\pi)}$; we aim to show that $F|_{L_I(\pi)} \models D$, which means that $F \models \bigvee_{l \in L_I(\pi)} \neg l \vee D$. Let $D^* \in F$ such that $D = D^*|_{L(\pi)}$. Notice that $D \neq \top$, so D^* does not include any of the literals l_{k_j} in $L_O(\pi)$.

   Suppose $\neg l_{k_j} \in D^*$ for some $l_{k_j} \in L_O(\pi)$. As l_{k_j} covers $l_{k_{j-1}}$, if D^* were in the set $\text{RC}(F, C', l_{k_j})$ for some $C' \subseteq C_{\text{EXT}}$, then $l_{k_{j-1}} \in D^*$. As $l_{k_j} \in L(\pi)$ for each $0 \leq j \leq p$, this contradicts $D \neq \top$.

   Otherwise, D^* must be excluded from the the set $\text{RC}(F, C', l_{k_j})$ because there is some $\neg l \in D$ for some $l \in C'$. If that literal l were in C, or added by covered literal addition, then $\neg l \in L(\pi)$ so that $D = \top$. If instead l was added by asymmetric literal addition, this means $F|_{L_I(\pi)} \models \neg l$, and thus $F \wedge \bigwedge_{l \in L_I(\pi)} \models D$.

   In case D^* includes none of the literals in $L_O(\pi)$ or the negations of any literals in $L_O(\pi)$, then D also occurs in $F|_{L_I(\pi)}$, thus the implication holds.

2. There is no such k_0, which means $L_I(\pi) = \{\neg l \mid l \in C_{\text{EXT}}\}$. Then π includes the edge (v_n, u_b), or the edge (v_{n-1}, v_n) if $b = n$, and similar to the previous case, π includes only edges from output vertices from there to the terminal. This means $L_O(\pi) = \{l_b, l_{k_1}, \ldots, l_{k_p}\}$, where l_{k_1} covers l_b, and each l_{k_j} covers $l_{k_{j-1}}$, for $1 \leq j \leq p$.

Again we aim to show that $F|_{L_I(\pi)} \models D$ for $D \in F|_{L(\pi)}$. Let $D^* \in F$ such that $D = D^*|_{L(\pi)}$. As before, $D \neq \top$ so D^* does not include any of the literals in $L_O(\pi)$. Further, if $\neg l_{k_j} \in D^*$ for some $l_{k_j} \in L_O(\pi)$ then the implication holds by the same argument as before.

Suppose instead $\neg l_b \in D^*$. First, notice that if C_{EXT} were subsumed in F then the implication holds, as $F|_{L_I(\pi)} \equiv \bot$. This is because $L_I(\pi)$ includes the negation of each literal in C and the negation of each covered literal in C_{EXT}, and further, for each l_i added to C_{EXT} as an asymmetric literal, $F|_{L_I(\pi)}$ includes the unit clauses $(\neg l_i)$. Therefore if C_{EXT} is subsumed then $F|_{L_I(\pi)} \equiv \bot$. We assume then l_b blocks C_{EXT} in F, so that $\neg l_b \in D^*$ means D^* includes the negation $\neg l$ of another literal in $l \in C_{\text{EXT}}$. If l was added to C_{EXT} as an asymmetric literal, then $F|_{L_I(\pi)} \models \neg l$, and thus $F \wedge \bigwedge_{l \in L_I(\pi)} \models D$.

If l' was not added as an asymmetric literal, and $l' \notin L_O(\pi)$, then $\neg l' \in L_I(\pi)$ and moreover $\neg l' \in L(\pi)$; this is a contradiction as $D \neq \top$. If instead $l' \in L_O(\pi)$ then $l' = l_{k_j}$ for some j, meaning $l_{k_{j-1}} \in D$, which is again a contradiction, as this would mean $D = \top$.

□

The theorem below is then a direct consequence of these two lemmas.

Theorem 4.5.1. *If a clause C is an asymmetric covered clause in a formula F, then the transformation diagram $G(C, C_{EXT}, b)$ is a witness for C with respect to F.*

Finally we consider the size of the graph $G = G(C, C_{\text{EXT}}, b)$. If the extended clause C_{EXT} includes N literals, then the graph G includes at most $2N$ vertices, each of which has at most two outgoing edges. Thus G can be described in space linear in N, for instance by listing all edges in G.

4.6 Conclusion

This chapter presented transformation diagrams, directed acyclic graphs for defining transformations, and demonstrated how they can be applied to assignments and composed with Boolean functions and formulas. We showed how transformation diagrams can be used as redundancy witnesses in an intuitive way, both for checking the redundancy of clauses and for solution reconstruction. Importantly, compact transformation diagram witnesses for covered clauses were described and proven correct.

The use of transformation diagrams in practice for performing solution reconstruction is yet to be evaluated. Another direction for future work is the use of

transformation diagrams for simplifying sequences of redundant clause additions or eliminations. The diagrams constructed for covered clauses can be seen as a condensed form of the standard reconstruction sequence using partial assignment witnesses for covered literal additions, exploiting the similarities shared between steps in this sequence. There may be other sequences of redundant clauses for which the witnesses can be simplified in a similar way. Further it may be possible to generalize this process, providing a way to compress reconstruction sequences, or possibly even proofs in systems based on redundant clause addition.

Chapter 5

Conclusion

This thesis presented an in-depth exploration of the limits of redundancy-based reasoning methods in SAT, and advanced approaches to redundancy to overcome these limits. In this final chapter, we review the contributions included in this thesis and discuss potential directions for future work.

5.1 Contributions

The contributions of this thesis are summarized below, separated by the chapter in which they are presented.

Chapter 2: Covered Clauses Are Not Propagation Redundant

This work revisits covered clause elimination, a clause elimination procedure based on a strong redundancy property. As this procedure had previously been only described mathematically, it provides and proves correct an explicit algorithm for deciding if a clause is asymmetrically covered with respect to a formula, and for reconstructing solutions to formulas after removing asymmetric covered clauses. Unlike for other redundancy properties, the reconstruction process utilized by this procedure does not produce partial assignment witnesses for the removed clauses, but uses a multi-step process for incrementally restoring solutions to the removed clauses. This is more burdensome with respect to the space required for solution reconstruction, but we prove that, even though it can be efficiently decided whether a clause is covered, a partial assignment witness for a covered clause can be as difficult to find as a satisfying assignment to an arbitrary formula.

As an consequence of this result we show covered clauses are not necessarily PR clauses, despite the generality of the PR redundancy property. Finally, we consider the complexity of clause redundancy itself. In particular, we show that deciding whether a clause is redundant with respect to a formula is a complete problem for the complement of the complexity class D^{P}, which was originally defined to classify problems that are hard for both NP and co-NP, but are seemingly not complete for either.

Chapter 3: Non-Clausal Redundancy Properties

In this chapter, we extend the ideas about clause redundancy, used in the design of formula simplification procedures and proof systems for SAT, to characterize redundancy of Boolean functions in general. We show that this general form of redundancy provides a framework for defining specific non-clausal redundancy properties, and in particular devise properties for the redundancy of functions expressed by Binary Decision Diagrams (BDDs). The properties $\mathsf{PR}_{\mathrm{BDD}}$ and $\mathsf{RUP}_{\mathrm{BDD}}$, defined in this chapter, are shown to easily express Gaussian elimination reasoning, an efficient SAT solving technique capable of quickly solving formulas that encode exclusive-or (XOR) constraints Importantly, this shows the strength of these properties over their clausal analogs: proof systems based on $\mathsf{PR}_{\mathrm{BDD}}$ and $\mathsf{RUP}_{\mathrm{BDD}}$ can make any inferences possible in systems based on PR and RUP, respectively, but it is not easy to express XOR reasoning using PR and RUP.

This chapter also presents the results of a preliminary implementation of a tool for checking proofs in systems based on BDD redundancy properties. The tool dxddcheck uses the $\mathsf{RUP}_{\mathrm{BDD}}$ property to verify proofs based on sequentially adding XOR constraints to a formula, and was evaluated on proofs derived from Gaussian elimination reasoning performed by the SAT solver Lingeling. The results demonstrate that dxddcheck can efficiently verify such proofs, and importantly that these proofs are significantly shorter than proofs for the same problems which do not use XOR reasoning. This demonstrates the potential of using non-clausal, BDD redundancy properties for proofs of unsatisfiability more generally.

Chapter 4: Redundancy by Transformation

This chapter investigates the use of transformations, functions mapping assignments to assignments, as witnesses for redundant clauses. Transformations, as introduced in Chapter 3, generalize the application of partial assignments to assignments, allowing for more flexible witnesses than are possible with partial assignments. However, they were used previously only in an abstract way, and it was unclear how to conveniently represent and work with transformations. We define directed acyclic graphs called transformation diagrams which describe transformations, and demonstrate how such a diagram can be used to compute the image of an assignment under the associated transformation. Further, we describe a way to compute an expression corresponding to the composition $f \circ \gamma$ of a Boolean function f by the transformation γ associated to a diagram, and show that this expression can easily be written in conjunctive normal form, given a conjunctive normal form for f.

We show how transformation diagrams can be used as witnesses for redundant clauses, and used in the reconstruction process employed by redundant clause elimination procedures. Finally, we consider the use of transformation diagrams as witnesses for covered clauses, and prove that any asymmetric covered clause has a transformation diagram witness which is linear in the size of the covering extension of that clause. As partial assignment witnesses for covered clauses may be hard to compute, and the current solution reconstruction process requires space quadratic in the size of the covering extension, this shows the advantage of transformation diagrams over partial assignment witnesses.

5.2 Future Work

There are a number of directions for extending and building upon the work presented in this thesis. In this final section, different lines of investigation are proposed for further advancing the usefulness of redundancy-based reasoning in SAT solving.

An important follow-up to the work presented on BDD redundancy properties is the development of the dxddcheck tool into a full implementation of a proof system based on the derivation of redundant BDDs. It is likely that an efficient implementation will need an optimized approach to carrying out the UnitProp procedure, which performs unit propagation over a collection of BDDs. Unit propagation over a set of clauses usually relies on watched-literal data structures [75] in order to avoid visiting many clauses which will not further propagation. It is clear that to perform UnitProp some BDDs can be skipped, for example by keeping track of which variables occur at which nodes, and comparing that with the variables currently assigned. However, this still involves visiting many BDDs which do not contribute to propagation, so it will be important to consider a way to extend watched literals, or a similar idea, to sets of BDDs.

The use of the $\mathsf{RUP}_{\mathrm{BDD}}$ and $\mathsf{PR}_{\mathrm{BDD}}$ redundancy properties for compressing clausal proofs should also be investigated. A sequence of clause additions, for example in a DRAT proof, could be expressed as a single BDD that is possibly more compact, especially if the clauses share multiple variables. If this BDD can be derived from the set of clauses by some BDD redundancy property, then the sequence of clause addition instructions could be replaced with an expression for the BDD. It will be important to evaluate to what extent this strategy can reduce the size of a proof. It is also clear that not all sequences of clauses, conjoined as a BDD, will be derivable by a BDD redundancy property. For example, a BDD for the conjunction of multiple clauses may include may include fewer variables than occur in the clauses, but these variables may have been necessary in the unit propagation checks that derived them, such as by RUP. As $\mathsf{RUP}_{\mathrm{BDD}}$ and

$\mathsf{PR}_{\mathrm{BDD}}$ also rely on unit propagation, this could prevent the BDD from being derivable by these properties.

Another direction is to design specific, novel formula simplification procedures which make use of the non-model-preserving capabilities of the $\mathsf{PR}_{\mathrm{BDD}}$ redundancy property. One idea is to consider other extensions of existing clausal procedures to BDDs; for example, is there a useful notion of a blocked BDD which is strictly more general than blocked clauses, but still lends itself to efficient identification? As formulas are typically presented as sets of clauses, a redundant BDD could indicate multiple clauses which can be removed, or added, at the same time. Alternatively, given strategies for efficiently working with both clauses and BDDs during solving, a separate set of learned BDDs could be managed and considered to be conjoined with the clause set.

There are a number of possible directions to explore for the use of transformation diagrams in solving as well, such as an implementation of the reconstruction procedure described in Chapter 4 which allows both partial assignments and transformation diagrams as witnesses for removed clauses. Such an implementation could be used to confirm experimentally that the transformation diagrams described for covered clauses are an efficient approach to reconstructing solutions to covered clauses. It could also be interesting to consider whether the stack of witness-labeled clauses, as used by the reconstruction function, could be compressed using transformation diagrams, possibly even replacing the entire stack with a single transformation diagram.

In general, the development of advanced notions of redundancy, such as those explored in this thesis, presents an opportunity for the investigation of possible new formula simplification strategies. Clause elimination procedures are typically designed to remove clauses which have an efficiently-identifiable witness; specifically, a partial assignment witness. It could be useful to develop, and evaluate experimentally, similar procedures which try to identify clauses, or sets of clauses, for which it can be efficiently determined that there is a corresponding witness in the form of a simple transformation.

Bibliography

[1] Ignasi Abío et al. "A New Look at BDDs for Pseudo-Boolean Constraints". In: *Journal of Artificial Intelligence Research* 45 (2012), pp. 443–480. DOI: 10.1613/jair.3653.

[2] M. Ajtai. "The Complexity of the Pigeonhole Principle". In: *Combinatorica* 14.4 (1994), pp. 417–433. DOI: 10.1007/BF01302964.

[3] Sheldon B. Akers. "Binary Decision Diagrams". In: *IEEE Transactions on Computers* 27.6 (1978), pp. 509–516. DOI: 10.1109/TC.1978.1675141.

[4] Gilles Audemard, George Katsirelos, and Laurent Simon. "A Restriction of Extended Resolution for Clause Learning SAT Solvers". In: *Proceedings of the 24th AAAI Conference on Artificial Intelligence* 24.1 (2010), pp. 15–20.

[5] John Backes et al. "Semantic-based Automated Reasoning for AWS Access Policies using SMT". In: *18th Conference on Formal Methods in Computer Aided Design – FMCAD*. Ed. by Nikolaj Björner and Arie Gurfinkel. 2018, pp. 206–214. DOI: 10.23919/FMCAD.2018.8602994.

[6] Tomás Balyo, Marijn J. H. Heule, and Matti Järvisalo. "SAT Competition 2016: Recent Developments". In: *31st AAAI Conference on Artificial Intelligence*. Ed. by Satinder P. Singh and Shaul Markovitch. AAAI Press, 2017, pp. 5061–5063.

[7] Lee A. Barnett and Armin Biere. "Non-clausal Redundancy Properties". In: *28th Intl. Conference on Automated Deduction – CADE*. Ed. by André Platzer and Geoff Sutcliffe. Vol. 12699. LNCS. Springer, 2021, pp. 252–272. DOI: 10.1007/978-3-030-79876-5_153.

[8] Lee A. Barnett, David Cerna, and Armin Biere. "Covered Clauses Are Not Propagation Redundant". In: *10th Intl. Joint Conference on Automated Reasoning – IJCAR*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Vol. 12166. LNCS. Springer, 2020, pp. 32–47. DOI: 10.1007/978-3-030-51074-9_3.

[9] Clark Barrett et al. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. IOS Press, 2021, pp. 1267–1329. DOI: 10.3233/FAIA201017.

[10] Roberto J. Bayardo and Robert Schrag. "Using CSP Look-Back Techniques to Solve Real-World SAT Instances". In: *14th AAAI National Conference on Artificial Intelligence*. Ed. by Benjamin Kuipers and Bonnie L. Webber. AAAI Press, 1997, pp. 203–208.

[11] Paul Beame, Henry Kautz, and Ashish Sabharwal. "Towards Understanding and Harnessing the Potential of Clause Learning". In: *Journal of Artificial Intelligence Research* 22.1 (2004), pp. 319–351. DOI: 10.1613/jair.1410.

[12] Armin Biere. "CaDiCaL at the SAT Race 2019". In: *Proceedings of SAT Race 2019*. Ed. by Marijn J. H. Heule, Matti Järvisalo, and Martin Suda. Department of Computer Science Series of Publications B. University of Helsinki, 2019, pp. 8–9.

[13] Armin Biere. "CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2018". In: *Proceedings of SAT Competition 2018*. Ed. by Marijn J. H. Heule, Matti Järvisalo, and Martin Suda. Department of Computer Science Series of Publications B. University of Helsinki, 2018, pp. 13–14.

[14] Armin Biere. "PicoSAT Essentials". In: *Journal on Satisfiability, Boolean Modeling and Computation* 4 (2008), pp. 75–97.

[15] Armin Biere, Matti Järvisalo, and Benjamin Kiesl. "Preprocessing in SAT Solving". In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. IOS Press, 2021, pp. 391–435. DOI: 10.3233/FAIA200992.

[16] Armin Biere et al. "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020". In: *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*. Ed. by Tomas Balyo et al. Vol. B-2020-1. Department of Computer Science Report Series B. University of Helsinki, 2020, pp. 51–53.

[17] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Transactions on Computers* 35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819.

[18] Randal E. Bryant and Marijn J. H. Heule. "Generating Extended Resolution Proofs with a BDD-Based SAT Solver". In: *27th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems – TACAS*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Vol. 12651. LNCS. Springer, 2021, pp. 76–93. DOI: 10.1007/978-3-030-72016-2_5.

[19] Jerry R. Burch et al. "Symbolic Model Checking for Sequential Circuit Verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.4 (1994), pp. 401–424. DOI: 10.1109/43.275352.

[20] Sam Buss and Neil Thapen. "DRAT Proofs, Propagation Redundancy, and Extended Resolution". In: *Theory and Applications of Satisfiability Testing – SAT 2019*. Ed. by Mikoláš Janota and Inês Lynce. Vol. 11628. LNCS. Springer, 2019, pp. 71–89. DOI: 10.1007/978-3-030-24258-9_5.

[21] Jin-Yi Cai and Lane Hemachandra. "The Boolean Hierarchy: Hardware over NP". In: *Structure in Complexity Theory*. Vol. 223. LNCS. Springer, 1986, pp. 105–124. DOI: 10.1007/3-540-16486-3_93.

[22] Philippe Chatalic and Laurent Simon. "Multi-Resolution on Compressed Sets of Clauses". In: *12th IEEE Intl. Conference on Tools with Artificial Intelligence – ICTAI*. IEEE Computer Society, 2000, pp. 2–10. DOI: 10.1109/TAI.2000.889839.

[23] Leroy Chew and Marijn J. H. Heule. "Sorting Parity Encodings by Reusing Variables". In: *23rd Intl. Conference on Theory and Applications of Satisfiability Testing – SAT*. Ed. by Luca Pulina and Martina Seidl. Vol. 12178. LNCS. Springer, 2020, pp. 1–10. DOI: 10.1007/978-3-030-51825-7_1.

[24] Edmund Clarke et al. "Bounded Model Checking Using Satisfiability Solving". In: *Formal Methods in System Design* 19.1 (2001), pp. 7–34. DOI: 10.1023/A:1011276507260.

[25] Stephen A. Cook. "The Complexity of Theorem Proving Procedures". In: *3rd Annual ACM Symposium on Theory of Computing – STOC*. ACM, 1971, pp. 151–158.

[26] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. "Verification of Synchronous Sequential Machines Based on Symbolic Execution". In: *Intl. Workshop on Automatic Verification Methods for Finite State Systems*. Ed. by Joseph Sifakis. Vol. 407. LNCS. Springer, 1990, pp. 365–373. DOI: 10.1007/3-540-52148-8_30.

[27] Olivier Coudert and Jean Christophe Madre. "A Unified Framework for the Formal Verification of Sequential Circuits". In: *IEEE Intl. Conference on Computer-Aided Design – ICCAD*. IEEE Computer Society, 1990, pp. 126–129. DOI: 10.1109/ICCAD.1990.129859.

[28] Olivier Coudert, Jean Christophe Madre, and Christian Berthet. "Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams". In: *2nd Intl. Workshop on Computer Aided Verification – CAV*. Ed. by Edmund M. Clarke and Robert P. Kurshan. Vol. 531. LNCS. Springer, 1990, pp. 23–32. DOI: 10.1007/BFb0023716.

[29] Robert F. Damiano and James H. Kukula. "Checking Satisfiability of a Conjunction of BDDs". In: *40th Design Automation Conference – DAC*. ACM, 2003, pp. 818–823. DOI: 10.1145/775832.776039.

[30] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. "A Scalable Algorithm for Minimal Unsatisfiable Core Extraction". In: *9th Intl. Conference on Theory and Applications of Satisfiability Testing - SAT*. Ed. by Armin Biere and Carla P. Gomes. Vol. 4121. LNCS. Springer, 2006, pp. 36–41. DOI: 10.1007/11814948_5.

[31] Niklas Eén and Armin Biere. "Effective Preprocessing in SAT Through Variable and Clause Elimination". In: *8th Intl. Conference on Theory and Applications of Satisfiability Testing – SAT*. Ed. by Fahiem Bacchus and Toby Walsh. Springer, 2005, pp. 61–75.

[32] Katalin Fazekas, Armin Biere, and Christoph Scholl. "Incremental Inprocessing in SAT Solving". In: *Theory and Applications of Satisfiability Testing – SAT 2019*. Ed. by Mikolás Janota and Inês Lynce. Vol. 11628. LNCS. Springer, 2019, pp. 136–154.

[33] John Franco et al. "SBSAT: A State-Based, BDD-Based Satisfiability Solver". In: *6th Intl. Conference on Theory and Applications of Satisfiability Testing – SAT*. Ed. by Enrico Giunchiglia and Armando Tacchella. Vol. 2919. LNCS. Springer, 2004, pp. 398–410. DOI: 10.1007/978-3-540-24605-3_30.

[34] Oliver Gableske. "BossLS Preprocessing and Stochastic Local Search". In: *Proceedings of SAT Challenge 2012*. Ed. by Adrian Balint et al. Department of Computer Science Series of Publications B. University of Helsinki, 2012, pp. 10–11.

[35] Vijay Ganesh and Moshe Y. Vardi. "On the Unreasonable Effectiveness of SAT Solvers". In: *Beyond the Worst-Case Analysis of Algorithms*. Ed. by Tim Roughgarden. Cambridge University Press, 2021, pp. 547–566. DOI: 10.1017/9781108637435.032.

[36] Allen Gelder. "Verifying RUP Proofs of Propositional Unsatisfiability." In: *10th Intl. Symposium on Artificial Intelligence and Mathematics – ISAIM*. 2008.

[37] Allen Van Gelder. "Extracting (Easily) Checkable Proofs from a Satisfiability Solver that Employs Both Preorder and Postorder Resolution". In: *7th Intl. Symposium on Artificial Intelligence and Mathematics – ISAIM*. 2002.

[38] Stephan Gocht and Jakob Nordström. "Certifying Parity Reasoning Efficiently Using Pseudo-Boolean Proofs". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.5 (2021), pp. 3768–3777.

[39] Evguenii I. Goldberg and Yakov Novikov. "Verification of Proofs of Unsatisfiability for CNF Formulas". In: *Conference on Design, Automation and Test in Europe– DATE*. IEEE Computer Society, 2003, pp. 886–891. DOI: 10.1109/DATE.2003.10008.

[40] Evguenii I. Goldberg, Mukul R. Prasad, and Robert K. Brayton. "Using SAT for Combinational Equivalence Checking". In: *Conference on Design, Automation and Test in Europe – DATE*. Ed. by Wolfgang Nebel and Ahmed Jerraya. IEEE Computer Society, 2001, pp. 114–121. DOI: 10.1109/DATE.2001.915010.

[41] Jan Friso Groote and Olga Tveretina. "Binary Decision Diagrams for First-Irder Predicate Logic". In: *Journal of Logic and Algebraic Programming* 57.1-2 (2003), pp. 1–22. DOI: 10.1016/S1567-8326(03)00039-0.

[42] Armin Haken. "The Intractability of Resolution". In: *Theoretical Computer Science* 39.2-3 (1985), pp. 297–308.

[43] Marijn Heule and Benjamin Kiesl. "The Potential of Interference-Based Proof Systems". In: *1st Intl. Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements – ARCADE*. Ed. by Giles Reger and Dmitriy Traytel. Vol. 51. EPiC Series in Computing. EasyChair, 2017, pp. 51–54.

[44] Marijn J. H. Heule. "Schur Number Five". In: *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*. 2018, pp. 6598–6606.

[45] Marijn J. H. Heule and Armin Biere. "All about Proofs, Proofs for All". In: vol. 55. Mathematical Logic and Foundations. College Publications, 2015. Chap. Proofs for Satisfiability Problems, pp. 1–22.

[46] Marijn J. H. Heule and Armin Biere. "What a Difference a Variable Makes". In: *Tools and Algorithms for the Construction and Analysis of Systems – TACAS*. Ed. by Dirk Beyer and Marieke Huisman. Vol. 10806. LNCS. Springer, 2018, pp. 75–92.

[47] Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. "Clause Elimination Procedures for CNF Formulas". In: *Logic for Programming, Artificial Intelligence, and Reasoning – LPAR 17*. Ed. by Christian G. Fermüller and Andrei Voronkov. Vol. 6397. LNCS. Springer, 2010, pp. 357–371.

[48] Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. "Covered Clause Elimination". In: *Logic for Programming, Artificial Intelligence and Reasoning – LPAR 17 (short paper)*. Ed. by Andrei Voronkov et al. Vol. 13. EPiC Series in Computing. EasyChair, 2010, pp. 41–46.

[49] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. "Clausal Proofs of Mutilated Chessboards". In: *11th NASA Formal Methods Symposium – NFM*. Ed. by Julia M. Badger and Kristin Yvonne Rozier. Vol. 11460. LNCS. Springer, 2019, pp. 204–210. DOI: 10.1007/978-3-030-20652-9_13.

[50] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. "Encoding Redundancy for Satisfaction-Driven Clause Learning". In: *25th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems – TACAS*. Ed. by Tomás Vojnar and Lijun Zhang. Vol. 11427. LNCS. Springer, 2019, pp. 41–58. DOI: 10.1007/978-3-030-17462-0_3.

[51] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. "Short Proofs Without New Variables". In: *Automated Deduction – CADE 26*. Ed. by Leonardo de Moura. Vol. 10395. LNCS. Springer, 2017, pp. 130–147. DOI: 10.1007/978-3-319-63046-5_9.

[52] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. "Strong Extension-Free Proof Systems". In: *Journal of Automated Reasoning* 64.3 (2020), pp. 533–554. DOI: 10.1007/s10817-019-09516-0.

[53] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. "Solving and Verifying the Boolean Pythagorean Triples Problem Cia Cube-and-Conquer". In: *Theory and Applications of Satisfiability Testing – SAT 2016*. Ed. by Nadia Creignou and Daniel Le Berre. Vol. 9710. LNCS. Springer, 2016, pp. 228–245. DOI: 10.1007/978-3-319-40970-2_15.

[54] Marijn J. H. Heule et al. "Clause Elimination for SAT and QSAT". In: *Journal of Artificial Intelligence Research* 53.1 (2015), pp. 127–168. ISSN: 1076-9757. DOI: 10.1613/jair.4694.

[55] Marijn J. H. Heule et al. "PRuning Through Satisfaction". In: *13th Haifa Verification Conference – HVC*. Ed. by Ofer Strichman and Rachel Tzoref-Brill. Vol. 10629. LNCS. Springer, 2017, pp. 179–194. DOI: 10.1007/978-3-319-70389-3_12.

[56] John N. Hooker. "Generalized Resolution and Cutting Planes". In: *Annals of Operations Research* 12 (1988), pp. 217–239. DOI: 10.1007/BF02186368.

[57] John N. Hooker. "Generalized Resolution for 0-1 Linear Inequalities". In: *Annals of Mathematics and Artificial Intelligence* 6 (1992), pp. 271–286. DOI: 10.1007/BF01531033.

[58] Kazuhisa Hosaka et al. "Size of Ordered Binary Decision Diagrams Representing Threshold Functions". In: *Theoretical Computer Science* 180.1-2 (1997), pp. 47–60. DOI: 10.1016/S0304-3975(97)83807-8.

[59] Matti Järvisalo, Armin Biere, and Marijn J. H. Heule. "Blocked Clause Elimination". In: *Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2010*. Ed. by Javier Esparza and Rupak Majumdar. Vol. 6015. LNCS. Springer, 2010, pp. 129–144. DOI: 10.1007/978-3-642-12002-2_10.

[60] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. "Inprocessing Rules". In: *International Joint Conference on Automated Reasoning – IJCAR*. Ed. by Bernhard Gramlich, Dale Miller, and Uli Sattler. Vol. 7364. LNCS. Springer, 2012, pp. 355–370. DOI: 10.1007/978-3-642-31365-3_28.

[61] Daher Kaiss et al. "Industrial Strength SAT-based Alignability Algorithm for Hardware Equivalence Verification". In: *7th Intl. Conference on Formal Methods in Computer Aided Design – FMCAD*. IEEE Computer Society, 2007, pp. 20–26. DOI: 10.1109/FAMCAD.2007.37.

[62] Henry A. Kautz and Bart Selman. "Planning as Satisfiability". In: *10th European Conference on Artificial Intelligence – ECAI*. John Wiley and Sons, Inc., 1992, pp. 359–363.

[63] Benjamin Kiesl, Marijn J. H. Heule, and Armin Biere. "Truth Assignments as Conditional Autarkies". In: *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*. Ed. by Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza. Vol. 11781. LNCS. Springer, 2019, pp. 48–64.

[64] Benjamin Kiesl et al. "Super-Blocked Clauses". In: *International Joint Conference on Automated Reasoning – IJCAR*. Ed. by Nicola Olivetti and Ashish Tiwari. Vol. 9706. LNCS. Springer, 2016, pp. 45–61. DOI: 10.1007/978-3-319-40229-1_5.

[65] Boris Konev and Alexei Lisitsa. "Computer-aided Proof of Erdős Discrepancy Properties". In: *Artificial Intelligence* 224 (2015), pp. 103–118. ISSN: 0004-3702. DOI: 10.1016/j.artint.2015.03.004.

[66] Andreas Kuehlmann and Florian Krohm. "Equivalence Checking Using Cuts and Heaps". In: *34th Design Automation Conference – DAC*. ACM, 1997, pp. 263–268. DOI: 10.1145/266021.266090.

[67] Oliver Kullmann. "On a Generalization of Extended Resolution". In: *Discrete Applied Mathematics* 96/97 (1999), pp. 149–176. DOI: 10.1016/S0166-218X(99)00037-2.

[68] C. Y. Lee. "Representation of Switching Circuits by Binary-Decision Programs". In: *The Bell System Technical Journal* 38.4 (1959), pp. 985–999.

[69] Leonid Levin. "Universal Search Problems". In: *Problemy Peredachi Informatsii* 9.3 (1973), pp. 115–116.

[70] Norbert Manthey. "Coprocessor 2.0 – a Flexible CNF Simplifier". In: *Theory and Applications of Satisfiability Testing – SAT 2012*. Ed. by Alessandro Cimatti and Roberto Sebastiani. Vol. 7317. LNCS. Springer, 2012, pp. 436–441. DOI: 10.1007/978-3-642-31612-8_34.

[71] Norbert Manthey, Marijn J. H. Heule, and Armin Biere. "Automated Reencoding of Boolean Formulas". In: *8th Haifa Verification Conference – HVC*. Ed. by Armin Biere, Amir Nahir, and Tanja Vos. Vol. 7857. LNCS. Springer, 2013, pp. 102–117. DOI: 10.1007/978-3-642-39611-3_14.

[72] João P. Marques-Silva and Karem A. Sakallah. "GRASP - A New Search Algorithm for Satisfiability". In: *IEEE Intl. Conference on Computer Aided Design – ICCAD*. IEEE, 1996, pp. 220–227. DOI: 10.1109/ICCAD.1996.569607.

[73] Fabio Massacci and Laura Marraro. "Logical Cryptanalysis as a SAT Problem". In: *Journal of Automated Reasoning* 24.1–2 (2000), pp. 165–203. DOI: 10.1023/A:1006326723002.

[74] Ilya Mironov and Lintao Zhang. "Applications of SAT Solvers to Cryptanalysis of Hash Functions". In: *Theory and Applications of Satisfiability Testing – SAT 2006*. Ed. by Armin Biere and Carla P. Gomes. Springer, 2006, pp. 102–115.

[75] M.W. Moskewicz et al. "Chaff: Engineering an Efficient SAT Solver". In: *38th Design Automation Conference – DAC*. ACM, 2001, pp. 530–535.

[76] DoRon B. Motter and Igor L. Markov. "A Compressed Breadth-First Search for Satisfiability". In: *4th Intl. Workshop on Algorithm Engineering and Experiments – ALENEX*. Ed. by David M. Mount and Clifford Stein. Vol. 2409. LNCS. Springer, 2002, pp. 29–42. DOI: 10.1007/3-540-45643-0_3.

[77] Oswaldo Olivo and E. Allen Emerson. "A More Efficient BDD-Based QBF Solver". In: *17th Intl. Conference on Principles and Practice of Constraint Programming – CP*. Ed. by Jimmy Lee. LNCS. Springer, 2011, pp. 675–690. DOI: 10.1007/978-3-642-23786-7_51.

[78] Guoqiang Pan and Moshe Y. Vardi. "Search vs. Symbolic Techniques in Satisfiability Solving". In: *7th Intl. Conference on Theory and Applications of Satisfiability Testing – SAT*. Vol. 3542. LNCS. Springer, 2004, pp. 235–250. DOI: 10.1007/11527695_19.

[79] Christos Papadimitriou and Mihalis Yannakakis. "The Complexity of Facets (and Some Facets of Complexity)". In: *Journal of Computer and System Sciences* 28.2 (1984), pp. 244–259. DOI: 10.1016/0022-0000(84)90068-0.

[80] Tobias Philipp and Adrian Rebola-Pardo. "DRAT Proofs for XOR Reasoning". In: *15th European Conference on Logics in Artificial Intelligence – JELIA*. Ed. by Loizos Michael and Antonis C. Kakas. Vol. 10021. LNCS. 2016, pp. 415–429. DOI: 10.1007/978-3-319-48758-8_27.

[81] Joachim Posegga and Bertram Ludäscher. "Towards First-order Deduction Based on Shannon Graphs". In: *16th German Conference on Artificial Intelligence – GWAI*. Ed. by Hans Jürgen Ohlbach. Vol. 671. LNCS. Springer, 1992, pp. 67–75. DOI: 10.1007/BFb0018993.

[82] Jussi Rintanen. "Planning and SAT". In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. IOS Press, 2021, pp. 765–789. DOI: 10.3233/FAIA201003.

[83] J. A. Robinson. "A Machine-Oriented Logic Based on the Resolution Principle". In: *Journal of the ACM* 12.1 (1965), pp. 23–41. DOI: 10.1145/321250.321253.

[84] Olivier Roussel and Vasco Manquinho. "Pseudo-Boolean and Cardinality Constraints". In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. IOS Press, 2021, pp. 1087–1129. DOI: 10.3233/FAIA201012.

[85] Detlef Sieling and Ingo Wegener. "Reduction of OBDDs in Linear Time". In: *Information Processing Letters* 48.3 (1993), pp. 139–144. DOI: 10.1016/0020-0190(93)90256-9.

[86] Carsten Sinz and Armin Biere. "Extended Resolution Proofs for Conjoining BDDs". In: *1st Intl. Computer Science Symposium in Russia – CSR*. Ed. by Dima Grigoriev, John Harrison, and Edward A. Hirsch. Vol. 3967. LNCS. Springer, 2006, pp. 600–611. DOI: 10.1007/11753728_60.

[87] Artur Soboń, Miroslaw Kurkowski, and Sylwia Stachowiak. "Complete SAT-based Cryptanalysis of RC5 Cipher". In: *Journal of Information and Organizational Sciences* 44.2 (2020), pp. 365–382. DOI: 10.31341/jios.44.2.10.

[88] Mate Soos, Stephan Gocht, and Kuldeep S. Meel. "Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling". In: *32nd Intl. Conference on Computer Aided Verification – CAV*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12224. LNCS. Springer, 2020, pp. 463–484. DOI: 10.1007/978-3-030-53288-8_22.

[89] Mate Soos, Karsten Nohl, and Claude Castelluccia. "Extending SAT Solvers to Cryptographic Problems". In: *12th Intl. Conference on Theory and Applications of Satisfiability Testing – SAT*. Ed. by Oliver Kullmann. LNCS. Springer, 2009, pp. 244–257. DOI: 10.1007/978-3-642-02777-2_24.

[90] Niklas Sörensson and Armin Biere. "Minimizing Learned Clauses". In: *12th Intl. Conference on Theory and Applications of Satisfiability Testing – SAT*. Ed. by Oliver Kullman. Vol. 5584. LNCS. Springer, 2009, pp. 237–243. DOI: 10.1007/978-3-642-02777-2_23.

[91] Hervé J. Touati et al. "Implicit State Enumeration of Finite State Machines Using BDDs." In: *IEEE Intl. Conference on Computer-Aided Design – ICCAD*. IEEE Computer Society, 1990, pp. 130–133. DOI: 10.1109/ICCAD.1990.129860.

[92] Grigorii Samuilovich Tseitin. "On the Complexity of Derivation in Propositional Calculus". In: *Studies in Constructive Mathematics and Mathematical Logic*. Ed. by Anatolii Olesevich Slissenko. Vol. 2. Steklov Mathematical Institute, 1970, pp. 115–125.

[93] Alasdair Urquhart. "Hard Examples for Resolution". In: *Journal of the ACM* 34.1 (1987), pp. 209–219. DOI: 10.1145/7531.8928.

[94] Alasdair Urquhart. "The Complexity of Propositional Proofs". In: *Bulletin of Symbolic Logic* 1.4 (1995), pp. 425–467. DOI: 10.2307/421131.

[95] Yakir Vizel, Vadim Ryvchin, and Alexander Nadel. "Efficient Generation of Small Interpolants in CNF". In: *25th Intl. Conference on Computer Aided Verification – CAV*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. LNCS. Springer, 2013, pp. 330–346. DOI: 10.1007/978-3-642-39799-8_23.

[96] Andrei Voronkov. "AVATAR: The Architecture for First-Order Theorem Provers". In: *26th Intl. Conference on Computer Aided Verification – CAV*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. LNCS. Springer, 2014, pp. 696–710. DOI: 10.1007/978-3-319-08867-9_46.

[97] J. P. Warners et al. "A Two Phase Algorithm for Solving a Class of Hard Satisfiability Problems". In: *Operations Research Letters* 23 (1998), pp. 81–88. DOI: 10.1016/S0167-6377(98)00052-2.

[98] Gerd Wechsung. "On the Boolean Closure of NP". In: *Intl. Conference on Fundamentals of Computation Theory – FCT*. Ed. by Lothar Budach. Vol. 199. LNCS. Springer, 1985, pp. 485–493. DOI: 10.1007/BFb0028832.

[99] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. "DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs". In: *17th Intl. Conference on Theory and Applications of Satisfiability Testing – SAT*. Ed. by Carsten Sinz and Uwe Egly. Vol. 8561. LNCS. Springer, 2014, pp. 422–429. DOI: 10.1007/978-3-319-09284-3_31.

[100] Lintao Zhang and Sharad Malik. "Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications". In: *Conference on Design, Automation and Test in Europe – DATE*. IEEE Computer Society, 2003, pp. 10880–10885. DOI: 10.1109/DATE.2003.10014.