Combinations of Boolean Gröbner Bases and SAT Solvers

Thanh Hung Nguyen

1. Gutachter: **Prof. Dr. Gerhard Pfister**
2. Gutachter: **Prof. Dr. Martin Kreuzer**

Datum der Disputation: 12.12.2014

D386

To my family

Acknowledgments

Contents

List of Algorithms

List of Figures

List of Tables

x

Introduction

In this thesis, we will combine Gröbner basis with SAT Solver in different manners. The *Boolean satisfiability (SAT) problem* is the problem of finding an assignment of a set of Boolean variables V such that a Boolean formula $F(V)$ will have the value *true* under this assignment. A tool that can solve a SAT problem is called a SAT Solver. The SAT problem is very intersting not only in theory but also in practice. Although the complexity of the SAT problem is NP-complete, many modern SAT solvers can still solve many real world problems efficiently, including hardware verification, software verification, planning and scheduling.

On the other hand, *Gröbner basis* is one of the central concepts in computer algebra. It is a specific generating set of an ideal over a polynomial ring with extremely nice properties. Gröbner basis techniques have many applications in algebraic geometry, optimization, coding, robotics, control theory, molecular biology, and many other fields.

Both SAT solvers and Gröbner basis techniques have their own strength and weakness. Combining them could fix their weakness. Subsequently, several research groups made considerable efforts to combine Gröbner basis techniques with SAT Solvers. Matthew Clegg at al. [CEI96] introduced a *Gröbner proof system* combining Buchberger's algorithm [Buc85] with the backtracking technique of a SAT Solver. In contrast, Condrat and Kalla used Gröbner basis to preprocess CNF formulæ [CK07] before giving them to a SAT solver, while Zengler and Küchlin [ZK10] use Gröbner basis techniques to obtain additional informations for SAT Solvers.

SAT solvers and Boolean Gröbner basis computations use different inputs. Most of the SAT solvers use proposition formulæ in conjunctive normal form (CNF) as input. In conjunctive normal form, only three operators, *and* ($\wedge$), *or* ($\vee$) and *negation* ($\neg$) are used to represent a propositional formula. The input for Boolean Gröbner basis computations is a system of Boolean polynomials. Boolean polynomial is a polynomial in $\mathbb{Z}_2[x_1, \ldots, x_n]$, where each term has degree at most one per variable. Therefore, to combine SAT Solvers and Gröbner basis techniques, we need an algorithm to convert Boolean polynomials to CNFs and another one to convert CNFs to Boolean polynomials. In this thesis, we will present some classical conversions as well as some refinements of these conversions.

The first combination is using Gröbner techniques to learn additional binary clauses for CDCL SAT solver from a selection of clauses. This combination is first proposed by Zengler and Küchlin [ZK10]. However, in our experiments, about 80 percent Gröbner basis computations give no new binary clauses. By selecting smaller and more compact input for Gröbner basis computations, we can significantly reduce the number of inefficient Gröbner basis computations, learn much more binary clauses. In addition, the new strategy can reduce the solving time of a SAT Solver in general, especially for large and hard problems.

The second combination is using all-solution SAT solver (find all satisfying assignments for given formula) and interpolation to compute Boolean Gröbner bases of Boolean elimination ideals of a given ideal. Computing Boolean Gröbner basis of the given ideal is the classical method to compute Boolean elimination ideals. However, this method is inefficient in case we want to eliminate most of the variables from a big system of Boolean polynomials. Therefore, we propose a more efficient approach to handle such cases. In this approach, we combine

all-solutions SAT Solver and interpolation to compute a Boolean Gröbner basis of elimination ideals. This means that we do not compute a Gröbner basis of the given ideal but only a Gröbner basis of the ideal corresponding to the projection given by elimination of some variables. The given ideal is translated to the CNF formula. Then an all-solution SAT Solver is used to find a finite set of points, the projection of all solutions of the given ideal. Finally, an algorithm, e.g. Buchberger-Moeller Algorithm, is used to associate the reduced Gröbner basis to this set of points. We also optimize the Buchberger-Moeller Algorithm for lexicographical ordering and compare it with the algorithm from [BD13], an alternative to the Buchberger-Moeller Algorithm for Boolean polynomials.

Finally, we combine Gröbner basis and abstraction techniques to the verification of some digital designs that contain complicated data paths, such as multiplier and filter. Firstly, abstraction techniques are used to lift bit operations to word operations whenever possible. By this way, we can construct an abstract model. Then, we reformulate it as a system of polynomials in the ring $\mathbb{Z}_{2^k}[x_1, \ldots, x_n]$. The variables are ordered in a way such that the system has already been a Gröbner basis w.r.t lexicographical monomial ordering. Finally, the normal form is employed to prove the desired properties. To evaluate our approach, we verify the global property of a multiplier and a FIR filter using the computer algebra system SINGULAR [DGPS12]. The result shows that our approach is much faster than the commercial verification tool from Onespin [One] on these benchmarks.

We structure the thesis as follows. Basic knowledge about SAT Solvers and Gröbner bases is given in chapter 1. In chapter 2, we discuss about the relations between Boolean polynomials (the input for Boolean Gröbner basis computations) and conjunctive normal forms (the input for SAT Solvers). Chapter 3 shows how to learn additional binary clauses for SAT solvers using Boolean Gröbner bases efficiently. Using all-solution SAT solver and interpolation to compute a Boolean Gröbner basis of elimination ideals is presented in chapter 4. In the final chapter, we combine Gröbner basis and abstraction techniques to verify digital designs containing complicated data paths.

Chapter 1

Boolean Satisfiability and Boolean Gröbner bases

1.1 Boolean Satisfiability

The Boolean Satisfiability problem is very intersting not only in theory but also in practice. It is the first problem to be proven NP-complete. However, modern SAT solvers can still solve many real world problems efficiently, including hardware verification, software verification, planning and scheduling.

Some classical notions as well as algorithms for state-of-the-art solving of the satisfiability problem will be given in this section, for more details see – for instance – [BHvMW09].

1.1.1 The satisfiability problem

The *Boolean satisfiability (SAT) problem* is the problem of finding an assignment ν of a set of Boolean variables V such that a given Boolean formula $F(V)$ will have the value *true* under this assignment. Then ν is called a *satisfying assignment*, or a *solution*, for F. The formula F is called *satisfiable* if it has a solution, otherwise F is called *unsatisfiable*.

The Boolean formula used in SAT problem usually in *conjunctive normal form* (CNF).

Definition 1.1.1. The Boolean formula is said to be in conjunctive normal form if it is a conjunction of clauses, while each *clause* is a disjunction of literals over V, and each *literal l* is an instance of a variable or its negation.

For convenience, we represent a CNF formula as a set of clauses and each clause as a set of literals. In addition, we also use literals to represent variable assignment. The assignment $x = true$ is denoted by literal x, and $x = false$ by literal $\neg x$.

By definition of CNF, a clause C is satisfied under an assignment ν if and only if C has at least one literal which is *true* under the assignment, and a CNF formula is satisfied if and only if all of its clauses are satisfied.

Definition 1.1.2. We call C a *conflict clause* if all literals of C are false, a *unit clause* if C has only one literal, a *binary clause* if C consists of two literals.

Example 1.1.3. $F = (\neg x \vee y) \wedge (x \vee \neg y \vee z)$ is a CNF formula. This CNF consists of 2 clauses

$$C_1 := \neg x \vee y \quad \text{and} \quad C_2 := x \vee \neg y \vee z$$

and 3 variables x, y, z. The clause C_1 consists of 2 literals $\neg x$ and y. The CNF can be rewritten as

$$F = \{\{\neg x, y\}, \{x, \neg y, z\}\}.$$

and the assignment $\nu = \{x, y, \neg z\}$, e.g $x = true, y = true, z = false$, is a solution for F. Therefore, F is satisfiable.

We can easily determine if a Boolean formula of 3 or 4 variables is satisfiable or unsatisfiable by testing all the possible variable assignments. However, Boolean formulæ in pratice are very large. They could have several thousands variables and millions of clauses. Therefore, the testing approach is infeasible in practice, a more efficient approach is in demand. A tool that can solve the SAT problems is called *SAT Solver*.

1.1.2 DPLL SAT Solver

Most of SAT Solvers are based on the *DPLL Algorithm*. It is a decision procedure proposed by Davis, Logemann and Loveland [DLL62] in 1962. This procedure refined the *DP Algorithm* proposed by Davis and Putnam [DP60] in 1960. Therefore, it is called the DPLL Algorithm.

The DP algorithm make use of the *resolution rule* while the DPLL algorithm bases on the *conditioning operation* and *existential quantification*.

Definition 1.1.4. Two literals are said to be complements if one is the negation of the other. Let C_1 and C_2 be clauses containing complementary literals. Assume that

$$C_1 = x \vee l_1 \vee \cdots \vee l_{k_1}, \quad C_2 = \neg x \vee l'_1 \vee \cdots \vee l'_{k_2}$$

where x is a variable, $l_1, \ldots, l_{k_1}, l'_1, \ldots, l'_{k_2}$ are literals. The *the resolvent* of C_1 and C_2 is defined to be

$$C := l_1 \vee \ldots l_{k_1} \vee l'_1 \vee \cdots \vee l'_{k_2}.$$

The rule that replaces C_1 and C_2 by their resolvent C is called *the resolution rule*.

The resolution rule in propositional logic is a single valid inference rule that produces a new clause implied by two clauses containing complementary literals.

The result of conditioning a CNF formula F on a literal l is the CNF formula, denoted by $F \mid l$, which can be obtained from F by

- removing all clauses which contain literal l, and

- removing the literal $\neg l$ from all clauses containing $\neg l$.

In notation, $F \mid l = \{C - \{l\} \mid C \in F, l \in C\}$

The result of existentially quantifying a variable x from a formula F is denoted by $\exists x F$ and defined as follows:

$$\exists x F := F \mid x \vee F \mid \neg x.$$

The most important property of existential quantification is that F is satisfiable if and only if $\exists x F$ is satisfiable. Notice that $\exists x F$ has one variable less than F (if F involves x).

Whenever there exists a unit clause in the CNF F, we could simplify F by conditioning it on the only literal of that unit clause. We continue this process until the result CNF contains no unit clauses. This technique is called *unit resolution* (or *unit propagation*).

The Algorithm 1.1.1 represents the DPLL Algorithm in pseudocode. It will call at first the function UNIT-RESOLUTION. This function will return

- I, a list of literals that is either presented as a unit clause in F or is derived from F by unit resolution.

- G, a new CNF which results from conditioning F on literals in I.

Then, the algorithm will check for termination based on G.

- if $G = \{ \}$, i.e. all clauses in F are satisfied, it returns I as solution for F

- if $\{ \} \in G$, i.e F contains a conflict clause, it returns unsatisfiable.

In case the two termination conditions above fail, the algorithm will pick a literal l in G, and solves two SAT problems with the new CNFs $G \mid l$ and $G \mid \neg l$. If one of them are satisfiable, it will return a solution for F, otherwise it will return unsatisfiable.

Algorithm 1.1.1 DPLL algorithm, DPLL(F)

Input: a CNF formula F.
Output: a set of literals representing a solution for F or UNSATISFIABLE.
  1: $(I, G) = $ UNIT-RESOLUTION(F).
  2: **if** $G = \{ \}$ **then**
  3:      **Return** I
  4: **else if** $\{ \} \in G$ **then**
  5:      **Return** UNSATISFIABLE
  6: **else**
  7:      Choose a literal l in G
  8:      **if** $L = DPLL(G \mid l) \neq$ UNSATISFIABLE **then**
  9:          **Return** $L \cup I \cup \{l\}$
  10:      **else if** $L = DPLL(G \mid \neg l) \neq$ UNSATISFIABLE **then**
  11:          **Return** $L \cup I \cup \{\neg l\}$
  12:      **else**
  13:          **Return** UNSATISFIABLE
  14:      **end if**
  15: **end if**

Example 1.1.5. Consider the CNF

$$F = \{\{x, y, z\}, \{\neg x, \neg y\}, \{y, \neg z\}\}.$$

We will use the algorithm 1.1.1 to check the satisfiability of F. Initially, there is no unit clause in F. In addition, the formula F is neither empty nor containing empty set. Therefore, the algorithm takes a literal x, and conditions F on x:

$$F \mid x = \{\{\neg y\}, \{y, \neg z\}\}.$$

Now, DPLL is applied again to check satisfiability of $F \mid x$. The formula $F \mid x$ contains a unit clause $\{\neg y\}$. Hence, the function UNIT-RESOLUTION($F \mid x$) is called, and return $I = \{\neg y, \neg z\}$ and $G = \{ \}$. The $DPLL(F \mid x)$ will return $\{\neg y, \neg z\}$ since G is empty. As a result, the $DPLL(F)$ will return $\{x, \neg y, \neg z\}$ which is a solution for F, that means F is satisfiable.

Among all DPLL-based SAT Solvers, the Conflict Driven Clause Learning (CDCL) SAT solver is the most efficient one. It can solve a lot of practical SAT problems. The power of the CDCL Solver comes from the ability of learning from conflicts. Some famous CDCL SAT Solvers are MiniSat [ES04], Glucose [AS09] and CryptoMiniSat [SNC09].

1.1.3 Conflict Driven Clause Learning SAT Solver

Algorithm 1.1.2 shows the standard organization of a Conflict Driven Clause Learning SAT solver. With respect to DPLL, the main differences are the call to function CONFLICTANAL-YSIS each time a conflict is identified, and the call to BACKTRACK when backtracking takes place. Moreover, the BACKTRACK procedure allows for backtracking non-chronologically. The following functions are used in algorithm 1.1.2:

1. UNITPROPAGATION consists of the iterated application of the *unit clause rule.* If all literals in a clause are *false* except one literal l, the literal l must be *true* to make the clause satisfy. This rule is called unit clause rule. If an unsatisfied clause is identified during unit propagation, then a conflict indication is returned.

2. PICKBRANCHINGVARIABLE consists of selecting a variable to assign and the respective value.

3. CONFLICTANALYSIS consists of analyzing the most recent conflict and learning a new clause from the conflict.

4. BACKTRACK backtracks to the decision level computed by CONFLICTANALYSIS.

5. ALLVARIABLESASSIGNED tests whether all variables have been assigned, in which case the algorithm terminates indicating that the CNF formula is satisfiable.

In the conflict analysis step, there are several learning schemes used to learn new clauses. To understand these learning schemes, we need the implication graph. This graph reflects the unit propagation process and conflicts (if they exists).

Definition 1.1.6. The *implication graph G* at a given state of DPLL is a directed acyclic graph with edges labeled with sets of clauses. It is constructed as follows:

1. Create a node for each decision literal, labeled with that literal.

2. While there exists a clause $C = \{l_1, \ldots, l_k, l\}$ such that all nodes with lables $\neg l_1, \ldots, \neg l_k$ are in G, but neither l nor $\neg l$ labels a node in G,

   (a) add a node with label l to G, and
   (b) add directed edges from l_i to l with label C, $1 \leq i \leq k$.

3. While there exists a clause $C = \{l_1, \ldots, l_k\}$ such that all nodes with labels $\neg l_1, \ldots, \neg l_k$ are in G,

   (a) add a node with label K to G, and
   (b) add directed edges from l_i to K with label C, $1 \leq i \leq k$.

The graph G contains a conflict if it contains a node K. Note that an implication graph can contain no conflict or one conflict or several conflicts. The Figure 1.1.3 shows 2 examples of implication graph.

In the rest of this section, we only consider the implication graph at a stage where there is at least one conflict and fix a conflict if there are several ones. Pick any cut in the implication graph that has all decision variables on one side, called *reason side* and the conflict node K on the other side, called *conflict side*. All nodes on the reason side that has an edge going to

the conflict side form a *cause* of the conflict. Let C be the clause which contains the negations of literals labeling nodes in the cause. C is called the learned clause associated to the cut.

The *First Unit Implication Point* is one of the best learning schemes and commonly used in most of modern CDCL SAT solvers. A *unique implication points* (UIP) [MSSSS96] of an implication graph G is a node of G such that all the paths from latest decision node to the conflict node K go through it. The *first UIP* (or *1-UIP*) is the one that is closest to the conflict node K. The *1-UIP cut* is the cut of the implication graph such that all nodes reachable from the 1-UIP are on conflict side and the rest nodes are on reason side. The learned clause associated with 1-UIP cut is called *1-UIP clause*.

Modern CDCL SAT solvers also often use search restart techniques [GSK98, BMS00] to avoid getting too deep into spaces with no solutions. Search restart causes the algorithm to restart itself, all assignments are taken back, but learned clauses are preserved.

Algorithm 1.1.2 Typical CDCL algorithm, CDCL(F, ν)

1: **if** (UNITPROPAGATION(F, ν) == **CONFLICT**) **then**
2:     **Return UNSAT**
3: **end if**
4: $dl \leftarrow 0$ // Decision level
5: **while** (**not** ALLVARIABLESASSIGNED (F, ν)) **do**
6:     $(x, v) = $ PICKBRANCHINGVARIABLE (F, ν)   // Decide stage
7:     $dl \leftarrow dl + 1$ // Increment decision level
8:     $\nu \leftarrow \nu \cup \{x, v\}$
9:     **if** (UNITPROPAGATION(F, ν) == **CONFLICT**)   // Deduce stage **then**
10:         $\beta = $ CONFLICTANALYSIS(F, ν)   // Diagnose stage
11:         **if** ($\beta < 0$) **then**
12:             **Return UNSAT**
13:         **else**
14:             BACKTRACK(F, ν, β)
15:             $dl \leftarrow \beta$ // decrement decision level
16:         **end if**
17:     **end if**
18: **end while**

The following example will help the reader understand in details how CDCL solve a SAT problem.

Example 1.1.7. Let F be the CNF consisting of the following clauses $r_1 = \{x_1, x_2\}$, $r_2 = \{\neg x_1, \neg x_4, \neg x_5\}$, $r_3 = \{x_2, x_5\}$, $r_4 = \{\neg x_3, x_4\}$, $r_5 = \{x_3, x_4\}$, $r_6 = \{x_3, \neg x_4\}$, $r_7 = \{\neg x_4, x_5\}$. The Algorithm 1.1.2 solves this SAT problem as follows. At first, it performs unit propagation on F, but F has no unit clause, hence there is nothing to propagate. It sets decision level to zero and continues as in Table 1.1. The implied literals are new assignments obtained from the unit propagation process. The corresponding implication graph is on the left hand side of Figure 1.1.3. Conflict occurs at this level, hence the graph contains the node K. There are two UIPs in this graph. The first UIP is x_4 and the second UIP is x_3. The 1-UIP cut will generate the 1-UIP clause $r_8 = \{\neg x_1, \neg x_4\}$. Add r_8 to F as learned clause. The backtrack level is the maximal level of variable in the 1-UIP clause without 1-UIP. In this case, the backtrack level is 1 which is the level of the variable x_1. The algorithm backtrack to level 1, and clear all assignments x_2, x_3, x_4 and x_5.

Table 1.1: First decisions of CDCL Algorithm

Decision literal	decision level	implied literals	conflict
x_1	1	none	no
x_2	2	none	no
x_3	3	x_4, x_5	$\{\neg x_1, \neg x_4, \neg x_5\}$

Figure 1.1: Implication graphs with 1-UIP cuts
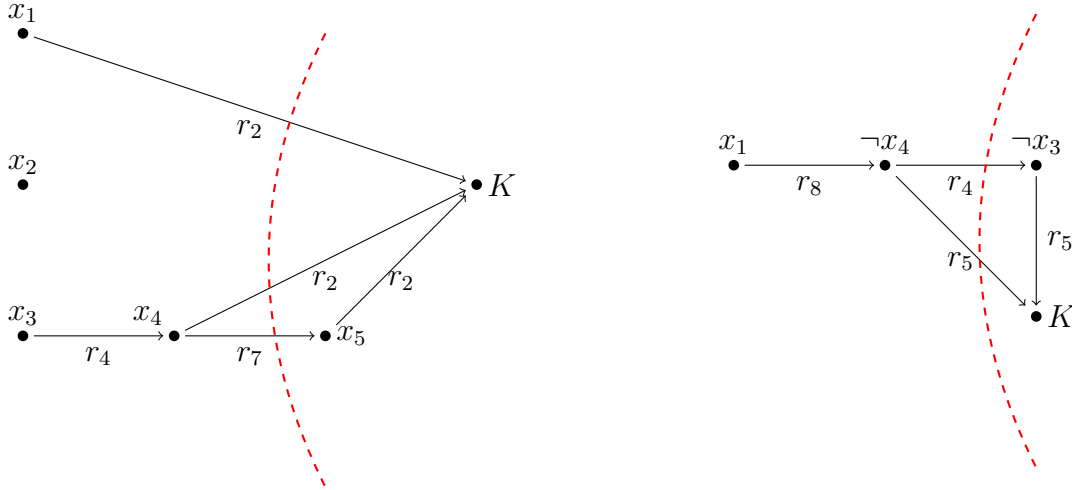
Now, the learned clause r_8 becomes an unit clause. Unit progation on F leads to another conflict. The implication graph on the right hand side of Figure 1.1.3 shows the situation of this conflict. Analyzing this conflict using 1-UIP learning scheme as before, the second learned clause $r_9 = \{x_4\}$ is added to F. The backtrack level for this conflict is 0. All assignment are clear. The CNF F has a unit clause $r_9 = \{x_4\}$. Perform unit propagation on F, the following assignments (literals) are implied: $x_4, x_3, x_5, \neg x_1$ and x_2. At this point, there is no conflict and all variable are assigned, hence the algorithm terminates and returns a solution of F:

$$\neg x_1, x_2, x_3, x_4, x_5.$$

We can see that all clauses in F are satisfied under these variable assignments.

1.2 Gröbner bases

In this section, we give some basic notions about Gröbner basis. For more details about Gröbner bases, the readers are referred to [zGG03, GP02].

A *Gröbner basis* is a specific generating set of an ideal over a polynomial ring. It has extremely nice properties and has many applications in algebraic geometry, optimization, coding, robotics, control theory, molecular biology, and many other fields. Bruno Buchberger (1965) invented the concept Gröbner bases, together with an algorithm to compute them, in his PhD thesis, and named it after his thesis advisor Wolfgang Gröbner. A Gröbner basis of an ideal depends on the *monomial ordering*, a special total ordering on the set of all monomials.

1.2.1 Monomial ordering

Let R be any principal ideal ring with 1. Let $R[\mathbf{x}] := R[x_1, \ldots, x_n]$ be the polynomial ring over R in variables $x_1, \ldots, x_n$. We denote

$$\mathbf{x}^\alpha := x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n}$$

for any $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{N}^n$.

Definition 1.2.1. We say $>$ is a monomial ordering on a polynomial ring $R[\mathbf{x}]$ if it is a total ordering on the set of monomials $Mon_n = \{\mathbf{x}^\alpha \mid \alpha \in \mathbb{N}^n\}$ satisfying

$$\mathbf{x}^\alpha > \mathbf{x}^\beta \implies \mathbf{x}^\gamma \mathbf{x}^\alpha > \mathbf{x}^\gamma \mathbf{x}^\beta$$

for all α, β, γ in $\mathbb{N}^n$.

We call a monomial ordering $>$ is global if $\mathbf{x}^\alpha > 1$ for all $\alpha \neq (0, \ldots, 0)$.

In this thesis, we only consider global monomial ordering. Examples for global orderings are the lexicographical ordering (denoted by lex), the degree lexicographical ordering (denoted by deglex). They are defined as follows:

$$\mathbf{x}^\alpha >_{lex} \mathbf{x}^\beta :\Longleftrightarrow \exists 1 \leq i \leq n : \alpha_1 = \beta_1, \ldots, \alpha_{i-1} = \beta_{i-1}, \text{ and } \alpha_i = \beta_i$$
$$\mathbf{x}^\alpha >_{deglex} \mathbf{x}^\beta :\Longleftrightarrow deg(\mathbf{x}^\alpha) > deg(\mathbf{x}^\beta)$$
$$\text{or } deg(\mathbf{x}^\alpha) = deg(\mathbf{x}^\beta) \text{ and } \mathbf{x}^\alpha >_{lex} \mathbf{x}^\beta$$

Some basic concepts in computer algebra are defined as follows:

Definition 1.2.2. Let $>$ be a fixed monomial ordering on $R[\mathbf{x}]$. Let $I \subset R[\mathbf{x}]$ be an ideal and $f \in R[\mathbf{x}]$ a non-zero polynomial. The polynomial f can be written in a unique way as sum of non-zero terms

$$f = a_\alpha \mathbf{x}^\alpha + a_\beta \mathbf{x}^\beta + \cdots + a_\gamma \mathbf{x}^\gamma$$

where $a_\alpha, a_\beta, a_\gamma \in R$ and $x^\alpha > x^\beta > \cdots > x^\gamma$. We defined

1. $LT(f) := a_\alpha \mathbf{x}^\alpha$, the leading term of f,

2. $LM(f) := \mathbf{x}^\alpha$, the leading monomial of f,

3. $LC(f) := a_\alpha$, the leading coefficient of f,

4. $tail(f) := f - LT(f)$, the tail of f,

5. $L(I) := \langle LT(f) \mid f \in I \rangle$, the leading ideal of I,

6. $LM(I) := \langle LM(f) \mid f \in I \rangle$, the leading monomial ideal of I

7. $V(I) := \{\mathbf{x} \in R^n \mid f(\mathbf{x}) = 0 \text{ for all } f \in I\}$, the common zeros or variety of I.

8. $I(V) := \{f \mid f(\mathbf{x}) = 0 \text{ for all } \mathbf{x} \in V\}$, the vanishing ideal of $V \subset R^n$

1.2.2 Normal forms and Gröbner bases

Normal forms of a polynomial with respect to a system of polynomials are defined as follows:

Definition 1.2.3. Let T denote the set of all finite subsets of $R[\mathbf{x}]$,

$$NF : R[\mathbf{x}] \times T \longrightarrow R[\mathbf{x}], \ (f, G) \mapsto NF(f, G)$$

is called a normal form on R if

1. $NF(0 \mid G) = 0$ for all $G \in T$,

2. $NF(f \mid G) \neq 0 \Rightarrow LT(NF(f \mid G)) \notin L(G)$ for all $f \in R$ and all $G \in T$,

3. If $G = \{g_1, \dots, g_s\}$, then, for any $f \in R$, $r := f - NF(f|G)$ has a standard representation with respect to G, that is,

$$r = f - NF(f \mid G) = \sum_{i=1}^{s} a_i g_i, a_i \in R, s \geq 0,$$

   satisfying $LM(r) \geq LM(a_i g_i)$ for all i such that $a_i g_i \neq 0$.

NF is called a reduced normal form if $NF(f|G)$ is reduced with respect to G, that is, leading terms of elements in G do not divide any terms of $NF(f \mid G)$.

The Algorithm 1.2.1, is called *Buchberger normal form*, computes a normal form of a polynomial with respect to $G \subset R[\mathbf{x}]$. The reduce normal form can be computed by the Algorithm 1.2.2.

The concept of s-polynomials used in the Algorithm 1.2.1 is defined as follows:

Definition 1.2.4. Let $f, g \in R[\mathbf{x}] \setminus \{0\}$. The s-polynomial of f and g is defined to be

$$spoly(f, g) := \frac{lcm(LT(f), LT(g))}{LT(f)} f - \frac{lcm(LT(f), LT(g))}{LT(g)} g$$

Algorithm 1.2.1 Normal form, $NF(f \mid G)$

Input: $f \in R[\mathbf{x}], G \subset R[\mathbf{x}]$, and a global monomial ordering $>$.
Output: $h \in R[\mathbf{x}]$, a normal form of f with respect to G.
  $h := f$
  while $h \neq 0$ and there exists a $g \in G$ such that $LT(g)$ divides $LT(h)$ **do**
    $h = spoly(h, g)$
  end while
  Return h

The polynomial ring $R[\mathbf{x}]$ is Noetherian, since R is Noetherian. Hence, every ideal in $R[\mathbf{x}]$ has a finite generating set. A Gröbner basis of an ideal is a particular generating set which has some extremely nice properties.

Definition 1.2.5. Let $I \subset R[\mathbf{x}]$ be an ideal. A finite set $G \subset R[\mathbf{x}]$ is called a Gröbner basis of I if

$$G \subset I, \text{ and } L(G) = L(I).$$

A Gröbner basis G of I is called reduced if $LC(f) = 1$ for any $f \in G$, and for any $f \neq g \in G$, $LM(g)$ does not divides any monomial of f.

Algorithm 1.2.2 Reduced normal form, $redNF(f \mid G)$

Input: $f \in R[\mathbf{x}], G \subset R[\mathbf{x}]$, and a global monomial ordering $>$.
Output: $h \in R[\mathbf{x}]$, a reduced normal form of f with respect to G.
  if $f = 0$ **then**
    Return f
  end if
  $f := NF(f \mid G)$
  Return $LT(f) + redNF(tail(f) \mid G)$

A Gröbner basis of an ideal in $R[\mathbf{x}]$ always exists, but the reduced Gröbner basic not necessarily. With a fixed monomial ordering, a Gröbner basis of an ideal I is still not unique, but the reduced Gröbner basis (if it exists) is unique.

Let I be an ideal in $K[\mathbf{x}]$, where K is a field, and let $>$ be a global ordering on $K[\mathbf{x}]$. Assume that I is generated by a finite set S. A Gröbner basis of I can be computed by the Buchberger Algorithm, the Algorithm 1.2.3.

Algorithm 1.2.3 Gröbner basis of $S \subset K[\mathbf{x}]$, K is field

---

Input: $S \subset K[\mathbf{x}]$, and $>$ a global ordering on $K[\mathbf{x}]$.
Output: $G \subset K[\mathbf{x}]$, a Gröbner basis of ideal generated by S in $K[\mathbf{x}]$.
  $G := S$
  $P := \{(f,g) \mid f,g \in G, f \neq g\}$, the pair-set.
  while $P \neq \emptyset$ **do**
    Choose $(f,g) \in P$
    $P := P \setminus \{(f,g)\}$
    $h := NF(spoly(f,g) \mid G)$
    if $h \neq 0$ **then**
      $P := P \cup \{(h,f) \mid f \in G\}$
      $G := G \cup \{h\}$
    end if
  end while
  Return G

Consider the case R being a principal ring (i.e every ideal in R can be generated by one element), e.g. $\mathbb{Z}_m := \mathbb{Z}/m\mathbb{Z}$. Gröbner basis computation over any principal ring is treated in [BDG$^+$09]. Here, we revise some basic notations

Definition 1.2.6. Let R be a principal ring and $a \in R$. The annihilator of a, $Ann(a) = \{b \in R \mid a \cdot b = 0\}$ is an ideal in R is generated by one element, which we denote by $NT(a)$.

Because of zero divisors, we need to extend the definition of an s-polynomial.

Definition 1.2.7. Let $f \in R \setminus \{0\}$. We define the *extended s-polynomial* of f to be

$$spoly(f,0) := spoly(0,f) := NT(LC(f)).f$$

The Algorithm 1.2.4 will return a Gröbner basis of an ideal in $R[\mathbf{x}]$, where R is a principal ring.

The following theorem ensures the correctness of the Algorithm 1.2.3 and 1.2.4. It is called Buchberger's criterion.

Algorithm 1.2.4 Gröbner basis of $S \subset R[\mathbf{x}]$, R is principal ring

Input: $S \subset R[\mathbf{x}]$, where R a principal ring, and $>$ a global ordering on $R[\mathbf{x}]$.
Output: $G \subset R[\mathbf{x}]$, a Gröbner basis of ideal generated by S in $R[\mathbf{x}]$.

$\quad G := S$
$\quad P := \{(f,g) \mid f,g \in G, f \neq g\} \cup \{(0,f) \mid f \in G\}$, the pair-set.
$\quad$**while** $P \neq \emptyset$ **do**
$\quad\quad$ Choose $(f,g) \in P$
$\quad\quad P := P \setminus \{(f,g)\}$
$\quad\quad h := NF(spoly(f,g) \mid G)$
$\quad\quad$**if** $h \neq 0$ **then**
$\quad\quad\quad P := P \cup \{(h,f) \mid f \in G\} \cup \{(0,h)\}$
$\quad\quad\quad G := G \cup \{h\}$
$\quad\quad$**end if**
$\quad$**end while**
$\quad$**Return** G

Theorem 1.2.8. *Let $G = \{f_0, f_1, \ldots, f_k\}$ be a set of generators of an ideal $I \subset R[\boldsymbol{x}]$ with $f_0 = 0$. If*

$$NF(spoly(f_i, f_j) \mid G) = 0 \text{ for } 0 \leq i \leq j \leq k$$

then G is a Gröbner basis of I.

1.3 Boolean Gröbner bases

We adopt Brickenstein's notations for Boolean Gröbner bases [Bri10].

1.3.1 Boolean polynomials

In this section, we consider the polynomial ring $\mathbb{Z}_2[\mathbf{x}] = \mathbb{Z}_2[x_1, \ldots, x_n]$. The equations $x^2 = x$ are always satisfied for every $x \in \mathbb{Z}_2$. Therefore, it is reasonable to consider the polynomials over $\mathbb{Z}_2$ modulo the so-called field polynomials

$$\mathbf{FP} = \{x_1^2 - x_1, x_2^2 - x_2, \ldots, x_n^2 - x_n\}$$

and obtain polynomials of degree at most 1 with respect to every variable.

Definition 1.3.1. A (multivariate) polynomial in $\mathbb{Z}_2[x_1, \ldots, x_n]$, such that each term has degree at most one with respect to every variable, is called a *Boolean polynomial*:

$$\deg_{x_i}(f) \leq 1 \qquad \forall i \in \{1, \ldots, n\}.$$

The set of all Boolean polynomials is denoted by $\mathbb{B}$.

Theorem 1.3.2. *The composition $\mathbb{B} \hookrightarrow \mathbb{Z}_2[\boldsymbol{x}] \twoheadrightarrow \mathbb{Z}_2[\boldsymbol{x}]/\langle \boldsymbol{FP} \rangle$ is a bijection. That is, the Boolean polynomials are a canonical system of representatives of residue classes in the quotient ring $\mathbb{Z}_2[\boldsymbol{x}]$ modulo the ideal generated by the field polynomials. Moreover, this bijection also provides $\mathbb{B}$ the structure of a $\mathbb{Z}_2$-algebra.*

Example 1.3.3. Let $f_1 = x_1 x_2 + x_3$ and $f_2 = x_3$ be two Boolean polynomials in $\mathbb{B} \simeq \mathbb{Z}_2[x_1, x_2, x_3]/\langle \mathbf{FP} \rangle$. Let f_4 and f_5 be the product and the sum of f_1 and f_2, respectively, then $f_4 = x_1 x_2 x_3 + x_3$ and $f_5 = x_1 x_2$ are also Boolean polynomials.

Definition 1.3.4. A function $F : \mathbb{Z}_2^n \longrightarrow \mathbb{Z}_2$ is called a Boolean function.

Theorem 1.3.5. *The map from $\mathbb{B}$ to the set of Boolean functions by mapping a polynomial to its polynomial function is an isomorphism of $\mathbb{Z}_2$-vector spaces.*

Proof. For a proof, we refer to [BDG$^+$09] □

Corollary 1.3.6. *Let p and q be two Boolean polynomials in $\mathbb{Z}_2[x_1, \ldots, x_n]$. If p and q have the same zero-set then $p = q$.*

Proof. Follows using Theorem 1.3.5. □

1.3.2 Boolean Gröbner bases

If G is a Gröbner basis of an ideal I with respect to some monomial ordering, then any $G' \subset I$ containing G is also a Gröbner basis of I. However, we can simplify a Gröbner basis to obtain a *reduced Gröbner basis.* The reduced Gröbner basis of an ideal is unique for a given monomial ordering. Therefore, it can be considered as a canonical representation of the ideal.

Definition 1.3.7. For any subset $H \subset \mathbb{Z}_2[\mathbf{x}]$, we define

$$\mathbf{BI}(H) := \langle H, \mathbf{FP} \rangle$$

the *Boolean ideal* of H. Let G be a finite set of Boolean polynomials in $\mathbb{Z}_2[\mathbf{x}]$. We call G a Boolean Gröbner basis of $\langle H \rangle$ if $G \cup \mathbf{FP}$ is a Gröbner basis of $\mathbf{BI}(H)$. We call G a reduced Boolean Gröbner basis of $\langle H \rangle$, $BGB(H)$ for short, if there exists a subset $S \subset \mathbf{FP}$ such that $G \cup S$ is a reduced Gröbner basis of $\mathbf{BI}(H)$.

By definition, the Buchberger algorithm can be used to compute the reduced Boolean Gröbner basis of an ideal in $\mathbb{Z}_2[\mathbf{x}]$. Besides the chain criterion and the product criterion, there is a new criterion for Boolean Gröbner bases. It is called Linear lead factor criterion which is stated in the following theorem.

Theorem 1.3.8. *Let $f \in \mathbb{B}$. If f can be written as $f = g.h$, where the leading monomial of g is x_i for some i, then the reduced normal form of $spoly(f, x_i^2 + x_i)$ with respect to $\{f\} \cup \boldsymbol{FP}$ is zero.*

Boolean polynomials manipulations as well as the reduced Gröbner basis of a Boolean ideal in $\mathbb{Z}_2[\mathbf{x}]$ can be computed efficiently in POLYBORI software developed by Brickenstein and Dreyer [BD09, Bri10]. See [Bri10] for more details.

Example 1.3.9. Let I be the ideal generated by $f_1 = x_1 x_2 + x_1$ and $f_2 = x_1 x_2 x_3$. Let $H := \langle \{f_1, f_2\} \cup \mathbf{FP} \rangle$ be the Boolean ideal of I. Consider the lexicographical monomial ordering with $x_1 > x_2 > x_3$. The reduced Gröbner basis H is $\{f_1, f_2, f_3\} \cup \mathbf{FP}$, where $f_3 = x_1 x_3$ By Definition 1.3.7, the reduced Boolean Gröbner basis of I is $\{f_1, f_2, f_3\}$.

Chapter 2

Relations between Boolean polynomials and CNFs

Formulæ in CNF are the main input for most SAT solvers, while Boolean Gröbner basis computation performs on Boolean polynomials. To cooperate the SAT solvers and Boolean Gröbner bases, we need a good conversion between Boolean polynomials and CNFs as well as understanding the relation between operations on these two objects.

2.1 Converting Boolean polynomials to CNFs

Let $\psi : \{0,1\} \longrightarrow \{True, False\}$ be the bijective map. The conversion must satisfy the condition $(x_1 = a_1, \ldots, x_n = a_n)$ fullfiles the equation $p = 0$ if and only if the assignment $x_1 = \psi(a_1), \ldots, x_n = \psi(a_n)$ satisfies the CNF of p.

Let ψ_1 be the map from the set of Boolean polynomials to the Boolean algebra that maps 1 to $True$ and 0 to $False$, and ψ_0 be an alternative of ψ_1, it maps 0 to $True$ and 1 to $False$. The following statements are true for all Boolean polynomials p and q.

1. $\psi_1(x + y) = \psi_1(x) \oplus \psi_1(y)$

2. $\psi_1(x \cdot y) = \psi_1(x) \wedge \psi_1(y)$

3. $\psi_0(x + y) = \psi_0(x) \wedge \psi_0(y)$

4. $\psi_0(x \cdot y) = \psi_0(x) \vee \psi_0(y)$.

Proof. These statements can be proven using truth tables. $\qquad\square$

Both maps can map Boolean polynomials to formulæ in Boolean logic. However, we need to convert Boolean polynomials to CNFs. Brickenstein has proposed a method to convert a Boolean polynomial to a CNF without introducing auxiliary variables. The CNF F associated with an ideal $I = \langle g_1, \ldots, g_m \rangle$ generated by Boolean polynomials, is the conjunction of CNFs associated with each Boolean polynomial in $\{g_1, \ldots, g_m\}$.

Let p be a Boolean polynomial and let O be the set of points in $\mathbb{Z}_2^n$ where the polynomial evaluates to one. The set O is so called *the one set* of p. The usual approach for CNF generation is finding prime blocks of O, which is defined as follows:

Definition 2.1.1. A set $B \subset V \subset \mathbb{Z}_2^n$ is called a block of V, if there exists sets $\emptyset \neq A_1, \ldots, A_n \subset \{0,1\}$, such that

$$B = \{(a_1, \ldots, a_n) \mid a_i \in A_i\}.$$

B is called a prime block of V, if B is not a proper subset of any block of V.

With the map ψ_1, each prime block $B = \{(a_1, \ldots, a_n) \mid a_i \in A_i\}$ of O will be associated with the clause that contains x_i if $A_i = \{0\}$, and $\neg x_i$ if $A_i = \{1\}$. With the map ψ_0, each prime block $B = \{(a_1, \ldots, a_n) \mid a_i \in A_i\}$ of O will be associated with the clause that contains x_i if $A_i = \{1\}$, and $\neg x_i$ if $A_i = \{0\}$. Conjunction of all clauses associated with all prime blocks forms a CNF of p.

The original CNF encoder in [Bri10] uses the map ψ_1. However, we prefer the map ψ_0. In the rest of this thesis, we only use the map ψ_0. The Algorithm 2.1.1 is a variant of the algorithm given in [Bri10]. It replaces ψ_1 by ψ_0.

Algorithm 2.1.1 ANF to CNF conversion

Input: p a Boolean polynomial
Output: F a CNF of p.
  $V := ones(p, \mathbb{Z}_2^n)$
  $T := V$
  $F := \emptyset$
  $i := 0$
  while $T \neq \emptyset$ **do**
    $i := i + 1$
    Choose $o \in T$
    $H := \{o\}$
    for $j \in \{1, \ldots, n\}$ **do**
      $c := \emptyset$ {Try to enlarge the set by adding the j-th unit vector to each element}
      $H' := \{h \mid h \in H\} \cup \{h + e_j \mid h \in H\}$
      if $H' \subset V$ **then**
        $H := H'$
      else
        if $o_j = 1$ **then**
          $c := c \cup \{x_j\}$
        else
          $c := c \cup \{\neg x_j\}$
        end if
      end if
      if $c \neq \emptyset$ **then**
        $F := F \cup \{c\}$
      end if
    end for
    $T := T \backslash H$
  end while
  Return F

Example 2.1.2. Let $p = x_1 + x_2 \cdot x_3$. To find the CNF of p, the algorithm 2.1.1 will find ones of p:

$$T := V := ones(p) = \{(0,1,1), (1,0,0), (1,0,1), (1,1,0)\}.$$

After that, it sets H as a block of V that contains only one point in T, e.g $(0,1,1)$, then tries to extend this block to a prime block of V by adding $j - th$ unit vector to each element of H. The following prime blocks give a cover of V:

$$H_1 = \{(0,1,1)\},$$

$$H_2 = \{(1, 0, 0), (1, 1, 0)\},$$
$$H_3 = \{(1, 0, 1), (1, 0, 0)\}.$$

The corresponding clauses are

$$c_1 = \{\neg x_1, x_2, x_3\}$$
$$c_2 = \{x_1, \neg x_3\}$$
$$c_3 = \{x_1, \neg x_2\}$$

The algorithm would return $F = \{c_1, c_2, c_3\}$ which is the CNF of p.

When there are several polynomials in I concerning the same set S of variables, we can convert them to a CNF at the same time. This conversion can further simplify the CNF of I. It can be done as follows: when converting a Boolean polynomial p to a CNF, we also looking for the group $SV(p)$ of Boolean polynomials in I containing the same set of variables as p. In the Algorithm 2.1.1, instead of setting $V := ones(p, \mathbb{Z}_2^n)$, we set

$$V := \bigcup_{q \in SV(p)} ones(q, \mathbb{Z}_2^n).$$

Example 2.1.3. Let $I = \{x_1 + x_2 + x_3, x_1 + x_2 \cdot x_3\}$ be a set of Boolean polynomials in $\mathbb{Z}_2[x_1, x_2, x_3]$. The one sets of polynomials in I are

$$ones(x_1 + x_2 + x_3) = \{(0, 0, 1), (0, 1, 0), (1, 0, 0), (1, 1, 1)\}$$
$$ones(x_1 + x_2 \cdot x_3) = \{(0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0)\}$$

Converting polynomials in I one by one will produce the CNF F of I containing following clauses:

$$\{\neg x_1, \neg x_2, x_3\}, \{\neg x_1, x_2, \neg x_3\}, \{x_1, \neg x_2, \neg x_3\},$$
$$\{x_1.x_2, x_3\}, \{\neg x_1, x_2, x_3\}, \{x_1, \neg x_3\}, \{x_1, \neg x_2\}.$$

However, polynomials in I having the same set of variables can be converted to a CNF at the same time. The resulting CNF of I in this case is $G = \{\{x_1\}, \{x_2\}, \{x_3\}\}$.

2.2 Converting CNFs to Boolean polynomials

Let $\phi : \{0, 1\} \longrightarrow \{True, False\}$ be the bijective map. The conversion must satisfy the condition $x_1 = a_1, \ldots, x_n = a_n$ satisfies the CNF F if and only if $(\phi(a_1), \ldots, \phi(a_n))$ is a zero of the Boolean polynomial of F.

Let ϕ_1 be the map from formulæ in propositional logic to Boolean polynomials mapping $True$ to 1 and $False$ to 0. Let ϕ_0 be an alternative of ϕ_1, mapping $True$ to 0 and $False$ to 1.

Lemma 2.2.1. *The following statements are true for every Boolean polynomials p and q.*

1. $\phi_1(p \vee q) = \phi_1(p) \cdot \phi_1(q) + \phi_1(p) + \phi_1(q)$

2. $\phi_1(p \wedge q) = \phi_1(p) \cdot \phi_1(q)$

3. $\phi_1(\neg p) = \phi_1(p) + 1$

4. $\phi_0(p \lor q) = \phi_0(p) \cdot \phi_0(q)$

5. $\phi_0(p \land q) = \phi_0(p) \cdot \phi_0(q) + \phi_0(p) + \phi_0(q)$

6. $\phi_0(\neg p) = \phi_0(p) + 1$.

Proof. These statements can be proven by truth tables. □

With one of the map ϕ_0 or ϕ_1, we can convert the whole CNF formula to a single Boolean polynomial, but usually the result is too complicated. For practical reasons, it is recommended to map each clause in the CNF to a Boolean polynomial and the whole CNF is mapped to a system of Boolean polynomials.

We prefer the map ϕ_0 since it is more natural to map $x_1 \lor x_2$ to $x_1 x_2$ than to $x_1 x_2 + x_1 + x_2$. In the rest of this thesis, we only use the map ϕ_0

Example 2.2.2. Let $F = C_1 \land C_2$ be a CNF formula, where $C_1 = x_1 \neg x_2$ and $C_2 = \neg x_3 \lor x_4$, then

$$\begin{aligned} f_1 := \phi_0(C_1) &= \phi_0(x_1) \cdot \phi_0(\neg x_2) \\ &= \phi_0(x_1) \cdot (\phi_0(x_2) + 1) \\ &= x_1(x_2 + 1) \\ &= x_1 x_2 + x_1 \end{aligned}$$

and, similarly, $f_2 := \phi_0(C_2) = x_3 x_4 + x_4$. Let $f_3 := \phi_0(C_1 \land C_2)$, then

$$f_3 = x_1 x_2 x_3 x_4 + x_1 x_2 x_3 + x_1 x_3 x_4 + x_1 x_2 + x_1 x_3 + x_3 x_4 + x_1 + x_3.$$

In practice, we would convert F to the system of Boolean polynomials $\{f_1, f_2\}$ instead of only one Boolean polynomials f_3.

Lemma 2.2.3. *Let C_1 and C_2 be clauses satisfying the condition: there exist a literal l in C_1 such that $\neg l$ is in C_2, then $\phi(C_1) \cdot \phi(C_2) = 0$*

Proof. By definition,

$$\phi_0(C_1) = \phi_0(C_1 \setminus \{l\}) \cdot \phi_0(l), \text{and } \phi_0(C_2) = \phi_0(C_1 \setminus \{l\}) \cdot (\phi_0(l) + 1).$$

Hence, the product of the two Boolean polynomials $\phi(C_1)$ and $\phi(C_2)$ is zero. □

Remark 2.2.4. It holds that $\psi(\phi(C)) = C$ for any clause C. However, the equality $\phi(\psi(p)) = p$ is not true for any Boolean polynomial p.

In case there exists a CNF sub-formula S containing clauses of the same variable, we propose converting S to one Boolean polynomial. This method can not only reduce the number of polynomials, but also reduce the degree of the polynomial. The following lemma allow simplify the conversion of the *and* operation in this case.

Lemma 2.2.5. *Let $S := \{C_1, \ldots, C_m\} \subset \{(l_1, \ldots, l_k) \mid l_i \in \{x_i, \neg x_i\}, i = 1, \ldots, k\}$ and assume that all C_i are pairwise different, then*

$$\phi_0(\bigwedge_{i=1}^{m} C_i) = \sum_{i=1}^{m} \phi_0(C_i).$$

Proof. By definition,

$$\phi_0(C_1 \wedge C_2) = \phi_0(C_1) \cdot \phi_0(C_2) + \phi_0(C_1) + \phi_0(C_2)$$
$$= (\phi_0(C_1) + 1)(\phi_0(C_2) + 1) + 1.$$

Therefore, by induction,

$$\phi_0(\bigwedge_{i=1}^{m} C_i) = \prod_{i=1}^{m} (\phi_0(C_i) + 1) + 1.$$

However, there always exists a literal l such that l in C_i and $\neg l$ in C_j for any $i \neq j$. By Lemma 2.2.3, $\phi_0(C_i) \cdot \phi_0(C_j) = 0$ for any $i \neq j$. Hence,

$$\phi_0(\bigwedge_{i=1}^{m} C_i) = \sum_{i=1}^{m} \phi_0(C_i).$$

$\square$

Example 2.2.6. The Algorithm 2.1.1 convert the Boolean polynomial $p = x_1 + x_2 + x_3$ to the CNF F consisting of following clauses

$$C_1 := \{\neg x_1, \neg x_2, x_3\}, C_2 := \{\neg x_1, x_2, \neg x_3\},$$
$$C_3 := \{x_1, \neg x_2, \neg x_3\}, C_4 := \{x_1.x_2, x_3\}\}.$$

Apply the map ϕ_0 to the whole formula F, the Boolean polynomial of F is computed as follows:

$$\phi_0(C_1) = x_1 x_2 x_3 + x_1 x_3 + x_2 x_3 + x_3,$$
$$\phi_0(C_2) = x_1 x_2 x_3 + x_1 x_2 + x_2 x_3 + x_2,$$
$$\phi_0(C_3) = x_1 x_2 x_3 + x_1 x_2 + x_1 x_3 + x_1,$$
$$\phi_0(C_4) = x_1 x_2 x_3,$$
$$\phi_0(F) = \phi_0(C_1) + \phi_0(C_2) + \phi_0(C_3) + \phi_0(C_4) = x_1 + x_2 + x_3.$$

Remark 2.2.7. With the new method, every linear polynomial will be recovered after forward and backward conversions.

2.3 Some relations between Boolean polynomials and CNFs

Lemma 2.3.1. *If we identify True with 0 and False with 1, then a clause C and its corresponding Boolean polynomial $\phi_0(C)$ have the same set of solutions. Therefore, the CNF formula $F = \{C_i \mid i \in I\}$ and the system of Boolean polynomials $J = \{\phi_0(C_i) \mid i \in I\}$ have the same solution set. Therefore, Boolean Gröbner basis techniques can solve SAT problems, but it is in general not a practical approach.*

Proof. By the identification and the property of conversion using the map ϕ_0, the clause C and the Boolean polynomial $\phi_0(C)$ have the same set of solutions. Hence

$$Solution(F) = \cap_{i \in I} Solution(C_i) = \cap_{i \in I} Solution(\phi_0(C)) = Solution(J).$$

$\square$

Lemma 2.3.2. *Let C be the non-trivial resolvent of two clauses, C_1 and C_2, then $\phi(C)$ is equal to the reduced normal form of the s-polynomial of $\phi(C_1)$ and $\phi(C_2)$ with respect to $\phi(C_1)$ and $\phi(C_2)$.*

Proof. The resolvent C is non-trivial, hence clause C_1 has exactly one literal l such that $\neg l$ is in C_2 We may assume that

$$C_1 = \bigvee_{i \in I_1} x_i \vee x_t \vee \bigvee_{j \in J_1} (x_j + 1)$$

$$C_2 = \bigvee_{i \in I_1} x_i \vee \neg x_t \vee \bigvee_{j \in J_1} (x_j + 1)$$

For convenience, we use the following notations

$$S_i = I_i \cup J_i \text{ for } i = 1, 2$$
$$S = S_1 \cup S_2$$
$$f_i = \phi(C_i) \text{ for } i = 1, 2$$
$$P_M = \prod_{m \in M} x_m \text{ for any finite index set } M$$

By definition of ϕ,

$$f_1 = \prod_{i \in I_1} x_i \cdot x_t \cdot \prod_{j \in J_1} (x_j + 1)$$

$$f_2 = \prod_{i \in I_2} x_i \cdot (x_t + 1) \cdot \prod_{j \in J_2} (x_j + 1)$$

Expand the last product in f_1 and f_2, and use above notations, we can rewrite f_1 and f_2 as follows

$$f_1 = P_{I_1} \cdot x_t \cdot \sum_{K_1 \subset J_1} P_{J_1 \setminus K_1} = \sum_{K_1} x_t P_{S_1 \setminus K_1}$$

$$f_2 = P_{I_2} \cdot (x_t + 1) \cdot \sum_{K_2 \subset J_2} P_{J_2 \setminus K_2} = \sum_{K_2} x_t P_{S_2 \setminus K_2} + \sum_{K_2} P_{S_2 \setminus K_2}$$

Here, K_i runs through the set of all subsets of J_i for $i = 1, 2$.

$$lead(f_1) = x_t P_{S_1}$$
$$lead(f_2) = x_t P_{S_1}.$$

Firstly, we prove lemma 2.3.2 for the case C_1 and C_2 having no common literals, i.e. $S_1 \cap S_2 = \emptyset$

$$spoly(f_1, f_2) = f_1 \cdot P_{S_2} + f_2 \cdot P_{S_1}$$
$$= \sum_{K_1} x_t P_{S \setminus K_1} + \sum_{K_2} x_t P_{S \setminus K_2} + \sum_{K_2} P_{S \setminus K_2}$$
$$= \sum_{K_1 \neq \emptyset} x_t P_{S \setminus K_1} + \sum_{K_2 \neq \emptyset} x_t P_{S \setminus K_2} + \sum_{K_2} P_{S \setminus K_2}.$$

We use following polynomial to reduce $spoly(f_1, f_2)$:

$$f = f_2 \cdot \sum_{K_1 \neq \emptyset} P_{S \setminus K_1} + f_1 \cdot \sum_{K_2 \neq \emptyset} P_{S \setminus K_2}$$

$$= \sum_{K_1 \neq \emptyset} \sum_{K_2} x_t P_{S \setminus (K_1 \cup K_2)} + \sum_{K_1 \neq \emptyset} \sum_{K_2} P_{S \setminus (K_1 \cup K_2)} + \sum_{K_1} \sum_{K_2 \neq \emptyset} x_t P_{S \setminus (K_1 \cup K_2)}$$

$$= \sum_{K_1 \neq \emptyset} x_t P_{S \setminus K_1} + \sum_{K_2 \neq \emptyset} x_t P_{S \setminus K_2} + \sum_{K_1} \sum_{K_2 \neq \emptyset} x_t P_{S \setminus (K_1 \cup K_2)}$$

Adding f to $spoly(f_1, f_2)$, we obtain

$$spoly(f_1, f_2) + f = \sum_{K_1} \sum_{K_2} P_{S \setminus (K_1 \cup K_2)}$$

$$= \sum_{K_1} P_{S_1 \setminus K_1} \sum_{K_2} P_{S_2 \setminus K_2}$$

$$= P_{I_1} \cdot P_{I_2} \sum_{K_1} P_{J_1 \setminus K_1} \sum_{K_2} P_{J_2 \setminus K_2}$$

$$= \prod_{i \in I_1 \cup I_2} x_i \cdot \prod_{j \in J_1 \cup J_2} (x_j + 1)$$

$$=: g$$

Both f_1 and f_2 have x_t in leading monomial, but no monomials in g contain x_t. Therefore, g is the reduced normal form of $spoly(f_1, f_2)$ w.r.t f_1 and f_2. Moreover, $\phi(C) = g$, so the lemma is proven for this case.

In the general case, C_1 and C_2 may share some common literals, i.e. $I_1 \cap I_2 \neq \emptyset$ and $J_1 \cap J_2 \neq \emptyset$.

Let

$$I := I_1 \cap I_2 \qquad\qquad I_i' := I_i \setminus I \text{ for } i = 1, 2$$
$$J := J_1 \cap J_2 \qquad\qquad J_i' := J_i \setminus J \text{ for } i = 1, 2$$

and

$$h := \prod_{i \in I} x_i \prod_{j \in J} (x_j + 1)$$

$$f_1' := \prod_{i \in I_1'} x_i \cdot x_t \cdot \prod_{j \in J_1'} (x_j + 1)$$

$$f_2' := \prod_{i \in I_2'} x_i \cdot (x_t + 1) \cdot \prod_{j \in J_2'} (x_j + 1)$$

then $f_1 = h \cdot f_1'$ and $f_2 = h \cdot f_2'$. Apply the previous result, the reduced normal form of $spoly(f_1', f_2')$ w.r.t f_1' and f_2' is

$$g' = \prod_{i \in I_1' \cup I_2'} x_i \cdot \prod_{j \in J_1' \cup J_2'} (x_j + 1).$$

Since h, f_1' and f_2' have no commom variables,

$$spoly(h \cdot f_1', h \cdot f_2') = h \cdot spoly(f_1', f_2')$$

It follows that

$$RedNF(spoly(f_1, f_2), \{f_1, f_2\}) = RedNF(h \cdot spoly(f_1', f_2'), \{h \cdot f_1', h \cdot f_2'\})$$

$$= h \cdot RedNF(spoly(f_1', f_2'), \{f_1', f_2'\})$$
$$= h \cdot g'$$
$$= \prod_{i \in I} x_i \prod_{j \in J} (x_j + 1)$$

$\square$

Chapter 3

Extending clause learning of SAT Solvers

We can use Boolean Gröbner bases to learn additional clauses for CDCL SAT Solvers. It is based on the following lemma.

Lemma 3.0.3. *Let F be a CNF and J be the ideal of F. Let C be a clause in the CNF of a Boolean polynomial $f \in \mathrm{BGB}(J)$, then F implies C, or C is a valid lemma of F.*

Proof. $V(C) \supset V(\phi^{-1}(f)) = V(f) \supset V(J) = V(F)$ where V(K) is standing for the solution set of K. This shows that the CNF formula $F \wedge C$ has the same set of solutions as F. $\qquad\square$

Example 3.0.4. Let $F = \{x_1 \vee \neg x_2,\ x_1 \vee x_2 \vee x_3$, then $J = \phi(F) = \{x_1 x_2 + x_1,\ x_1 x_2 x_3\}$. With the variable ordering $x_1 > x_2 > x_3$ and lexicographical monomial ordering,

$$\mathrm{BGB}(J) = \{x_1 x_4,\ x_1 x_3 + x_1,\ x_1 x_2 + x_1,\ x_2 x_3 + x_2\}$$

The newly generated Boolean polynomial is $x_1 x_3$. This polynomial corresponds to the clause $x_1 \vee x_3$. It is a valid lemma of F.

3.1 Extending clause learning of SAT Solvers

The principal idea was first proposed by Zengler and Küchlin in [ZK10]. We denote this original approach as ZK_GB. In conflict analysis, they perform learning as usual (following the 1-UIP strategy) and extend clause learning as follows:

1. Add all reason clauses of 2 to 8 literals, involved in the conflict to a set R.

2. If R has 4 to 6 clauses, then the corresponding system of polynomials is considered as a good input.

3. If the number of good inputs is a multiple of $2^{\#restarts}$, then they compute the BGB of the last good input.

4. Collect all new polynomials with 2 variables from BGB and add their corresponding clauses to the set of original clauses at next restart.

All parameters in their approach are chosen heuristically. The above parameters are not strong enough to control the quality of the selection. For example, it is impossible to deduce any binary clauses from a set of 6 clauses with length 8, see Lemma 3.1.1.

Lemma 3.1.1. *Let m and k be natural numbers and $m < k$, then there are no clauses of length $k - m$ which can be deduced from m clauses of length k.*

Proof. Let $F = \{C_1, \ldots, C_m\}$ be the set of m clauses of length k. Assume that we can deduce from F a clause of length $k - m$, say $l_{k-m+1} \vee l_{k-m+2} \vee \cdots \vee l_k$. Let x_i be the variable of the literal l_i for $i = k - m + 1, \cdots, k$.

On the other hand, we can always choose a variable x_i in C_i such that $x_i \notin S_i$, where $S_i = \{x_1, \ldots, x_{i-1}, x_{k-m+1}, \ldots, x_k\}$ for i from 1 to m, since C_i has k different variables and S_i has less than k variables. We assign to x_i a truth value such that C_i is evaluated to true for $i = 1, \ldots, m$. Therefore, ν together with $\{x_1, \ldots, x_m\}$ is a solution of F, contradictory to above assumption. $\qquad\square$

To fix this issue, we propose a completely new strategy. To increase possibilities of getting new binary clauses, clauses in the selection should be shorter and have more common variables. Therefore, we select clauses of length at most 4 instead of 8 and introduce a new parameter to adjust the number of common variables.

Definition 3.1.2. The *density of variables in a set of clauses* R is defined to be the quotient of the number of variables in R and the number clauses in R. We denote it by $dens(R)$.

Let m and n be the number of clauses and variables in the selection R, respectively. Let a and c be the average length of the clauses and the number of occurrences of variables in R, respectively. Let A be the average occurrences of a variable in R. Then,

$$A = \frac{c}{n} = \frac{a.m}{dens(R).m} = \frac{a}{dens(R)}.$$

The expression above implies that A is reversely proportional to $dens(R)$ if a is fixed. In particular, if u is the upper bound of $dens(R)$, then a/u will be the lower bound of A. When u is smaller, the selection has a better quality, but it is harder to find the selection R satisfying $dens(R) \leq u$.

Computing a Gröbner basis for each conflict is expensive. Therefore, we adopt the ZK_GB's strategy, the Gröbner basis is often computed at the beginning and seldom at the end.

Based on the above observation, we were able to improve Zengler and Küchlin's approach. In the same amount of time, our approach can learn much more binary clauses in each Gröbner basis computation as well as in total. Moreover, it can reduce the total SAT solving time in general.

We denote our approach by ND_GB which is defined as follows:

1. If the number of conflicts is a multiple of $2^{\#restarts}$, then extending binary clause learning by BGB, or Gröbner-learning for short, is activated.

2. When Gröbner-learning is activated, we collect at most 7 reason clauses such that each clause has at most 4 literals at conflict analysis and add them to a list R.

3. If R has at least 4 elements and the density of variables in R is at most 1.4, we convert every element in R to the corresponding polynomial, compute BGB of these polynomials, and deactivate the Gröbner-learning.

4. Only collect new polynomials with two variables and then add their corresponding clauses to the set of original clauses at next restart.

Table 3.1: Compare two BGB learning schemes.

name	k	ZK_GB				ND_GB			
		GBtime	BinCls	GBs	GBzeros	GBtime	BinCls	GBs	GBzeros
approve	17	5236	74373	194830	77%	323	80243	18859	6%
bioinfo	20	44517	56632	1346806	98%	444	124706	27355	30%
bitverif	11	17096	629918	1045119	69%	8291	4443896	558542	30%
c32sat	4	7271	11959	171503	96%	114	67474	6241	8%
crypto	11	493	24898	18106	52%	379	210077	23354	3%
palacios/uts	6	4540	67530	138423	72%	781	122631	60696	56%
parity-games	21	49437	378742	1563407	83%	1790	252054	145230	40%
sum	90	128590	1244052	4478194	84%	12122	5301081	840277	32%

3.2 Implementation and Benchmarks

In order to experiment with our hybrid Gröbner/CDCL-SAT approach, we added Gröbner
basis functionality to the MiniSat, a famous SAT Solver by Niklas Eén et al. [ES04]. Our
experiments showed that the approach of Zengler and Küchlin works better for version 070721
of the tool. We compute Gröbner bases using the PolyBoRi framework for computations
with Boolean polynomials of Brickenstein and Dreyer [BD09, BD13]. The latter was accessed
from MiniSat by embedding the programming language Python [RD06]. This allows for fully
accessing PolyBoRi's high-level algorithms and heuristics. All benchmarks are preprocessed
by MiniSat's SATElite-based preprocessor [EB05] before actually solving takes place.

Like Zengler and Küchlin we select a large set of benchmarks from the *SAT Competition
2009* [1] in the categories aprove09, bioinfo, bitverif, c32sat, crypto, palacios/uts and parity-
games. However, they select the subsets of examples computable with respect to a given time
out. In order to avoid such a bias towards our method we used a more generic choice: We take
all benchmarks (except minxor128 in bitverif[2]) in the sets of benchmarks that MiniSat 2.1
solved in the first phase of the main track of the competition.

For convenience, we will use some abbreviations in the tables below. GBtime is the time
spent for Boolean Gröbner basis computations. BinCls is the number of binary clauses learned
by Boolean Gröbner basis computations. GBs is the number of Boolean Gröbner basis com-
putations. GBzeros is the number of Boolean Gröbner basis computations that learn no new
binary clauses. Ratio is average number of binary clauses learned in one Boolean Gröbner
basis computation.

In order to compare the strengths and the weaknesses of our methods with Zengler and
Küchlin's approach, we used these benchmark examples for three different experiments.

First of all we evaluated the local efficiency of both clause selection strategies. For this
purpose, we computed the BGB of any set of clauses satisfying the criteria. To avoid affecting
on the search we do not add clauses learned during Gröbner basis computation to the SAT
solver. Since we throw away some useful results, this is a somewhat artificial setting. But it
allows directly comparing both methods in terms of clause selection.

Table 3.1 shows that our approach could learn more than four times the number of binary
clauses compared to ZK_GB. In addition, our approach just spends about 10% of the time

---

[1]See http://www.satcompetition.org/2009/.

[2]The benchmark minxor128 is easy for MiniSat 2.1, but it takes more than 2 days for MiniSat 070721 to
solve it.

Table 3.2: Analyze affects of density upper-bounds.

upper-bound	GBtime	BinCls	GBs	GBzeros	Ratio
1.20	4929	3339453	306172	35%	10.9
1.25	6703	3573694	425067	34%	8.4
1.30	10154	5093819	689707	30%	7.4
1.35	10799	5184031	737096	30%	7.0
1.40	12122	5301081	840277	32%	6.3
1.45	23496	7697614	1483807	28%	5.2
1.50	28409	7972266	1818357	34%	4.4

Table 3.3: Affects of density upper-bounds on solving time

upper-bound	Total time	GBtime	BinCls
1.25	55949	39	12715
1.30	64535	77	16812
1.35	46028	85	16633
1.40	23069	93	16348
1.45	43723	133	16752
1.50	34752	147	15925

ZK_GB spends for Gröbner basis computations. The reason is that we compute fewer Boolean Gröbner bases, but each of the Gröbner basis computation usually yields much more binary clauses. We outperform Zengler's and Küchlin's approach on learning new binary clauses by Boolean Gröbner bases.

In the second test, we want to motivate the density upper-bound setting. Therefore, we use the same settings as above, but we vary the density upper-bound. Table 3.2 shows that the larger density upper-bound is, the more binary clauses are learned in total, the more time we spent in computing Gröbner bases, but the less binary clauses are learned in a Gröbner computation. From these experiments the value of 1.4 seems to be a well-balanced setting.

To analyze the effect of our new parameter on solving time, we use our original setting, but vary the density upper bound. Table 3.3 shows that the density upper bound has a very strong effect on solving time. Again a density upper bound of 1.4 give the best score among the others.

Finally, we use the original setting of both approaches. The solving time of both approaches together with pure MiniSat is shown in Table 3.4. Our approach gets the best score among others in 4of 7 benchmark categories (bioinfo, crypto, palacios/uts and parity-games) and also in total solving time. The reason is that our solver can solve hard benchmarks faster, see Fig. 3.1. Table 3.5 shows that our solver can learn more than 15 times the number of binary clauses that Zenglin's and Küchlin's method can learn, but the time spent for computing Gröbner bases is almost the same.

3.3 Conclusion

By introducing the density of variables as a criterion for the clause selection of the Gröbner part, we significantly improved the efficiency of the hybrid Gröbner/CDCL-SAT approach.

Table 3.4: Compare solving time.

Name	Pure	ZK_GB	ND_GB
aprove09	2569	736	3060
bioinfo	6361	7088	6057
bitverif	4950	12801	5172
c32sat	798	811	1001
crypto	3210	4587	2702
palacios/uts	1339	1253	1155
parity-games	24794	13655	3922
sum	44021	40931	23069

Table 3.5: Analyze binary clause learned by BGBs.

name	ZK_GB				ND_GB			
	GBtime	BinCls	GBs	GBzeros	GBtime	BinCls	GBs	GBzeros
aprove09	11	265	365	68%	10	1917	650	12%
bioinfo	36	50	998	96%	24	1508	1310	68%
bitverif	13	228	512	67%	12	3760	751	29%
c32sat	9	5	184	98%	4	889	176	36%
crypto	2	174	133	51%	17	5091	805	9%
palacios	2	77	113	69%	12	1602	786	58%
parity	25	239	597	75%	14	1581	1021	42%
sum	98	1038	2902	80%	93	16348	5499	40%
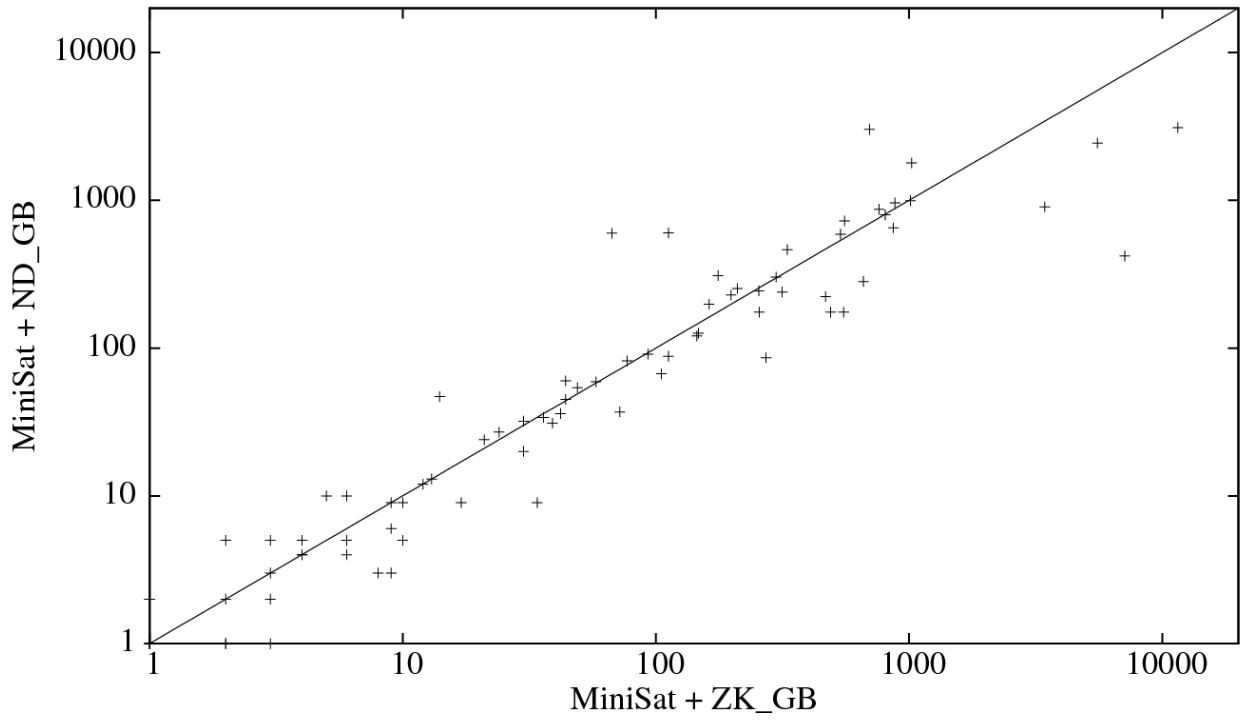
Figure 3.1: SAT solving time in seconds.

Using our heuristic we learn more binary clauses per computed Gröbner basis than the state-of-the-art approach of Zengler and Küchlin. In particular, the number of Gröbner basis computations yielding no useful information is reduced. This significantly improves the overall runtime of SAT solving for a crucial set of benchmark examples.

Chapter 4

Elimination by all-solutions SAT and interpolation

Boolean Gröbner bases is the classical tool to compute Boolean elimination ideals. However, this method is inefficient in case we want to eliminate most of the variables from a big system of Boolean polynomials. Therefore, we propose a more efficient approach to handle such cases. In this approach, we combine all-solutions SAT Solver and interpolation to compute a Boolean Gröbner basis of elimination ideals. This means that we do not compute a Gröbner basis of the given ideal but only a Gröbner basis of the ideal corresponding to the projection given by elimination of some variables. The given ideal is translated to the CNF formula. Then an all-solution SAT Solver is used to find a finite set of points, the projection of all solutions of the given ideal. Finally, an algorithm, e.g. Buchberger-Möller Algorithm, is used to associate the reduced Gröbner basis to this set of points. We also optimize the Buchberger-Möller Algorithm for lexicographical ordering and compare it with the algorithm from [BD13], an alternative to the Buchberger-Möller Algorithm for Boolean polynomials.

4.1 Gröbner bases and Elimination

We use the definition of an elimination ideal and the elimination theorem from [GP07]

Definition 4.1.1. Given $I = \langle f_1, \ldots, f_s \rangle \subseteq k[x_1, \ldots, x_n]$, the i-th **elimination ideal** of I defined by

$$I_i = I \cap k[x_{i+1}, \ldots, x_n] \tag{4.1}$$

An i-th elimination ideal does not contain the variables $x_1, \ldots, x_i$, neither does the basis generating it. The basis of an elimination ideal can be a Gröbner basis by using the elimination theorem:

Theorem 4.1.2. *Let $I \subseteq k[x_1, \ldots, x_n]$ be an ideal and G be a Gröbner basis of I with respect to a lex ordering where $x_1 \succ x_2 \succ \cdots \succ x_n$. Then for every $0 \le i \le n$, the set*

$$G_i = G \cap k[x_{i+1}, \ldots, x_n] \tag{4.2}$$

is a Gröbner basis of the i-th elimination ideal I_i.

The proof for elimination theorem can be found in [GP07], page 70.

Example 4.1.3. Consider the ideal I in $\mathbb{Z}_2[x, y, z, u, v]$ generated by

$$S = \{y + x + xu, v + y + yz)\}$$

To eliminate x, y, z, we need to re-order variables such that all eliminated variables are greater than the remaining variables, e.g

$$u \prec v \prec x \prec y \prec z$$

The reduced Boolean Gröbner basis of I w.r.t lexicographical ordering is

$$G = \{xzu + xz + xu + x + v, zv, y + x + xu, xv + v, uv\}$$

Then, by elimination theorem, $G \cap \mathbb{Z}_2[u, v] = \{uv\}$ is a Gröbner basis of $I \cap \mathbb{Z}_2[u, v]$.

4.2 All-SAT Problem

Given a Boolean formula presented in CNF, the All-SAT problem is the problem of finding all of its solutions, as defined in the SAT problem. The all-SAT problem has many applications in Artificial Intelligence and logic minimization.

The *Blocking Clauses Method* is a straight forward method to find all solutions of a formula. Whenever the DPLL SAT Solver finds a solution, add the blocking clause describing the negations of the solution to the solver, this can prevent the solver from reaching this solution again. The last decision is then invalidated and the search continues normally. Continue this proccess until no more solution is found and the algorithm will terminate.

For instances, SAT solver find the solution $x_i = True$ and $y_j = False$ for $i, j \in \{1, \ldots, 100\}$ to a formula F of 200 variables, then the following blocking clause will be added to the solver

$$\neg x_1 \vee \neg x_2 \vee \cdots \vee \neg x_{100} \vee y_1 \vee y_2 \vee \cdots \vee y_{100}.$$

There could be millions of solutions, then millions of such a long clauses will be added to the solver. Therefore, blocking clauses can blow up the memory and slow down the solvers.

Orna Grumberg at el. [GSY04] developed a memory-efficient All-SAT engine that can avoid adding blocking clauses. Therefore, the speed of the solver as well as the used memory is not affected by the number of solutions found. Moreover, this engine can find the set of all the assignments to a subset of variables, namely important variables, which can be extended to solutions of the formula.

Example 4.2.1. Use the CNF encoder, as explained in the first chapter, to convert the ideal in Example 4.1.3 to a CNF formula

$$F = \{y \vee \neg x \vee u, \neg y \vee x, \neg u \vee \neg y, y \vee \neg v, v \vee \neg y \vee z, \neg v \vee \neg z\}$$

All assignments to (u, v) that can be extended to a solution of F are

$$A = \{(0, 0), (0, 1), (1, 0)\}$$

This means that any extension of $(1, 1)$ can not satisfy F.

4.3 The Buchberger-Möller Algorithm for Boolean Polynomials

Vanishing ideals of a finite set of points are of interest not only in mathematics, like in coding theory and statistics, but also in molecular biology [LS04]. The Buchberger-Möller

Algorithm 4.3.1 Classical Buchberger-Möller Algorithm

Input: $P = \{P_1, ..., P_m\}$ a set of points in K^n and $\succ$ a monomial ordering on $K[x_1, \ldots, x_n]$

Output: The reduced Gröbner basis G of the vanishing ideal $I(P)$ and a set of standard monomials B

1: Initialization: $G = \emptyset, B = \emptyset, L = \{1\}$, and C a $0 \times m$ matrix
2: **while** $L \neq \emptyset$ **do**
3:     Set $t = min_\succ(L)$ and remove t from L
4:     Compute the evaluation vector $v = (t(P_1), \ldots, t(P_m)) \in K^n$
5:     **if** $v = \sum_j a_j r_j$, a linear combination of the rows of C **then**
6:         add $t - \sum_j a_j B[j]$ to G, and remove all multiples of t from L.
7:     **else**
8:         add v as a new row to C, append t to B,
9:         add to L those elements of $\{x_1 t, \ldots, x_n t\}$ which are neither multiples of an element of L nor of $lead(G)$.
10:     **end if**
11: **end while**
12: **Return** G, B

Algorithm (BMA, for short) was proposed in [MB82] as a tool to compute these vanishing ideals. The algorithm below is a formulation of the Classical Buchberger-Möller Algorithm as given in [ABKR00].

The main idea of the BM Algorithm is that it evaluates monomials one by one in increasing order (the smallest one is 1) at all points of P. Whenever the set of considered monomials can build up a polynomial f that vanish at all points of P, adding f to G as an element of the desired reduced Gröbner basis. Moreover, all multiples of the leading monomial of f will not be evaluated, this will ensure that G is reduced. In Algorithm 4.3.1, the list L will be updated m times where m is the number of points in P. Each time, the algorithm needs to check which monomials in $\{x_1 t, \ldots, x_n t\}$ are multiples of an element of L or of $lead(G)$. On the other hand, the algorithm has to find the smallest monomial in a set of monomials L at least m times. These two tasks are really time consuming. Therefore, we propose a new method to update L and find the minimal element of L in constant time in case of the lexicographical ordering.

Our method relates to the algorithm $SM - A$ in [JS06]. The algorithm $SM - A$ generates first a set of candidate standard monomials, then sorts them in increasing order. The paper [Lun08] shows that sorting monomials is a time consuming task. Our method generates a sorted list (w.r.t lex) of candidate standard monomials. Let $B_i^k := \{u \in B \cap K[x_i, \ldots, x_n] \mid deg_{x_i} u = k\}$ for $k \geq 1$. Let $B_i := B \cap K[x_i, \ldots, x_n]$. Let

$$L_i^1 = \{u \cdot x_i \mid u \in B_{i+1}\}$$
$$L_i^k = \{u \cdot x_i \mid u \in B_i^{k-1}\}$$

The Algorithm 4.3.3 will return $L = L_i^{k+1}$ if B_i^k is not empty. Otherwise, it will return $L = L_{i-1}^1$ if $i > 1$, and return empty list if $i = 1$.

Lemma 4.3.1. $B_i^k \subset L_i^k$.

Proof. Let v be an element of B_i^k, then v is in $B \cap K[x_i, \ldots, x_n]$ and $deg_{x_i} v = k$. We can write $v = x_i \cdot u$. Then, u is also a standard monomial of $I(P)$ since it is a divisor of a standard

monomial. Therefore, v is in B_{i+1} if $k = 1$ or in B_i^{k-1} if $k > 1$. This fact proves that $B_i^k \subset L_i^k$ for all $k \geq 1$. $\square$

Lemma 4.3.2. *The list L generated by the Algorithm 4.3.3 has following properties:*

1. *L is in increasing order, and*

2. *L contains no multiples of any element in $lead(G)$ for the current G.*

3. *monomials in L are smaller than any candidate standard monomials not in L.*

Proof. The first two properties are inherent from B and $Bnew$ in the Algorithm 4.3.3. The last one is true because of the lexicographical ordering. $\square$

Algorithm 4.3.2 Optimize BMA for lexicographical ordering

Input: $P = \{P_1, ..., P_m\}$ a set of points in K^n, X a list of variables in decreasing order $X_1 > X_2 > \cdots > X_n$.

Output: The reduced Gröbner basis G of the vanishing ideal $I(P)$ w.r.t lex. and the set of standard monomials B.

    Initialization: $G = \emptyset, B = \emptyset, Bnew = \emptyset, L = [1], i = n$ and C a $0 \times m$ matrix over K

    while $L \neq \emptyset$ **do**

        Set t is the first element of L, remove t from L.

        Compute the evaluation vector $v = (t(P_1), \ldots, t(P_m))$

        if $v = \sum_j a_j r_j$, a linear combination of the rows of C **then**

            add $t - \sum_j a_j B[j]$ to G, and remove all multiples of t from L.

        else

            add v as a new row to C, append t to the end of $Bnew$.

        end if

        if $L = [\,]$ **then**

            $L, i = gen\_cand\_stdmonos(B, Bnew, i)$

            Append $Bnew$ to the end of B

            $Bnew = \emptyset$

        end if

    end while

    Return G, B

Example 4.3.3. Consider $P = \{(0,0), (0,1), (1,0), (1,1)\} \subset \mathbb{Q}^2$, and the lexicographical ordering on $\mathbb{Q}[x, y]$ with $x > y$. The Table 4.1 and 4.2 show the steps of BMA and BMAlex applied to P, respectively. They yield the same result $G = \{y^2 - y, x^2 - x\}, B = \{1, y, x, xy\}$ in the same number of steps. However, the BMA needs more work than the BMAlex. The BMA has to identify the monomials in the column *check list* which are neither multiples of monomials in L nor in $lead(G)$, then use these monomials to extend L. In addition, the BMA has to find the minimal monomial of L while the first monomial of L in the BMAlex is already the minimal monomial of L.

    The update method for Boolean polynomials and lexicographical ordering is presented in the Algorithm 4.3.4.

Algorithm 4.3.3 *gen_sorted_cand_stdmonos*

Input: $B, Bnew, i$

Output: A list of candidate standard monomials

  if $Bnew = \emptyset$ **then**

    $i = i - 1$

    if $i > 0$ **then**

      $L = \{u \cdot X_i | u \in B\}$

    else

      $L = \emptyset$

    end if

  else

    $L = \{u \cdot X_i | u \in Bnew\}$

  end if

  Return L, i

Table 4.1: BMA example

Step	$t = min(L)$	$L - \{t\}$	Add to B	Add to G	Reduced L	Check list	Extended L
0		$\{1\}$					
1	1	$\emptyset$	1			x, y	$\{x, y\}$
2	y	$\{x\}$	y			xy, y^2	$\{x, y^2\}$
3	y^2	$\{x\}$		$y^2 - y$	$\{x\}$		
4	x	$\emptyset$	x			x^2, xy	$\{x^2, xy\}$
5	xy	$\{x^2\}$	xy			x^2y, xy^2	$\{x^2\}$
6	x^2	$\emptyset$		$x^2 - x$	$\emptyset$		

Table 4.2: BMAlex example

Step	$t = first(L)$	$L - \{t\}$	Add to B	Add to G	Reduced L	Check list	Extended L
0		$\{1\}$					
1	1	$\emptyset$	1				$\{y\}$
2	y	$\emptyset$	y				$\{y^2\}$
3	y^2	$\emptyset$		$y^2 - y$	$\emptyset$		$\{x, xy\}$
4	x	$\{xy\}$	x				
5	xy	$\emptyset$	xy				$\{x^2, x^2y\}$
6	x^2	$\{x^2y\}$		$x^2 - x$	$\emptyset$		

Algorithm 4.3.4 *gen_sorted_cand_stdmonos_bool*

Input: $B, Bnew, i$

Output: A sorted list of candidate Boolean standard monomials

  $i = i - 1$

  if $i > 0$ **then**

    $L = \{u \cdot X_i \mid u \in (B \cup Bnew)\}$

  else

    $L = \emptyset$

  end if

  Return L, i

4.4 Ideal of points by interpolation

In this section we revise the main algorithms in [BD13] which is used to compute the reduced lexicographical Gröbner basis of a variety. We also add some descriptions as well as examples to clarify the algorithms. The termination and correctness of the algorithms are proven in [BD13].

A partial Boolean function $f : \mathbb{Z}_2^n \to \mathbb{Z}_2$ can be defined by two disjoint subsets Z and O of $\mathbb{Z}_2^n$ where

- $f(x) = 0$ for every $x \in Z$,

- $f(x) = 1$ for every $x \in O$.

Therefore, we can denote f by b_Z^O. The domain of b_Z^O is $D = Z \cup O$.

We can define the sum of two partial functions as follows

$$(b_{Z_1}^{O_1} + b_{Z_2}^{O_2}) : \mathbb{Z}_2^n \longrightarrow \mathbb{Z}_2$$
$$x \longmapsto b_{Z_1}^{O_1}(x) + b_{Z_2}^{O_2}(x)$$

By this definition, the sum of two partial functions also a partial function

$$b_{Z_1}^{O_1} + b_{Z_2}^{O_2} = b_{(Z_1 \cap Z_2) \cup (O_1 \cap O_2)}^{(Z_1 \cap O_2) \cup (O_1 \cap Z_2)}$$

and its domain is

$$D = (Z_1 \cup O_1) \cap (Z_2 \cup O_2) = D_1 \cap D_2$$

Definition 4.4.1. Let $>$ be an arbitrary monomial ordering, then we can extend $>$ lexicographically to the set of Boolean polynomials (we make use the fact that all non-zero coefficients are one) by setting $p > q$ if and only if $p \neq 0$ and one of the following conditions holds

- $q = 0$

- $q \neq 0$ and $\mathrm{lm}(p) > \mathrm{lm}(q)$,

- $q \neq 0$, $\mathrm{lm}(p) = \mathrm{lm}(q)$ and $\mathrm{tail}(p) > \mathrm{tail}(q)$.

Lemma 4.4.2. *Let $I \supset \langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle$ be an ideal in $\mathbb{Z}_2[x_1, \ldots, x_n]$, p a Boolean polynomial, and let G be a Gröbner basis of I with respect to a global monomial ordering. Then the following two statements are equivalent:*

- *p is the lexicographically smallest Boolean polynomial in $p + I$*

- *p is reduced w.r.t G.*

This lemma is proven in [BD13].

Let K be a field, $X = \{P_1, \ldots, P_m\} \subset K^n$ be a finite set of points and $\phi : X \to K$ be a function on X, then there exist a polynomial $f \in K[x_1, \ldots, x_n]$ such that $f(P_i) = \phi(P_i)$ for $i = 1, \ldots, m$. This is consequence of the Chinese Remainder Theorem. By the Chinese remainder theorem, the map

$$K[x_1, \ldots, X_n]/I(X) \longrightarrow K^m$$

mapping every f to $(f(P_1), \ldots, f(P_m))$ is an isomorphism of rings. Therefore, for each i, there exists a polynomial h_i such that $h_i(P_i) = 1$ and $h_i(P_j) = 0$ for every $j \neq i$. The polynomial

$$f = \sum_{i=1}^{n} \phi(P_i) h_i,$$

satisfies $f(P_i) = \phi(P_i)$ for $i = 1, \ldots, m$.

Definition 4.4.3. A Boolean polynomial p is called an interpolation of a partial function f if $p(x) = f(x)$ for every $x \in P$, where $P = O \cup Z$.

Algorithm 4.4.1, named *interpolate_smallest_lex*, can compute the smallest interpolation polynomial w.r.t lexicographical monomial ordering under all polynomials interpolating the same function on P

Algorithm 4.4.1 *interpolate_smallest_lex*(b_Z^O)

Input: b_Z^O a partial function definition
Output: smallest Boolean polynomial p w.r.t. lex. and $f_p = b_Z^O$ on $Z \cup O$
  if $O = \emptyset$ **then**
    Return 0
  end if
  if $Z = \emptyset$ **then**
    Return 1
  end if
  $i = min(top(O), top(Z))$
  $Z_1 = subset1(Z, x_i)$
  $Z_0 = subset0(Z, x_i)$
  $O_1 = subset1(O, x_i)$
  $O_0 = subset0(O, x_i)$
  $C = (Z_1 \cup O_1) \cap (Z_0 \cup O_0)$
  $f = b_{Z_1}^{O_1}$
  $g = b_{Z_0}^{O_0}$
  $h_t = interpolate\_smallest\_lex(f + g)$
  $F = ones(h_t, (Z_1 \cup O_1) \backslash C)$
  $w = b_{((Z_1 \backslash C) \oplus F) \cup Z_0}^{((O_1 \backslash C) \oplus F) \cup O_0}$
  $h_e = interpolate\_smallest\_lex(w)$
  Return $x_i.h_t + h_e$

Example 4.4.4. Let $Z = \{(0,0)\}$ and $O = \{(0,1), (1,0)\}$. We will apply the algorithm 4.4.1 to find the smallest lex. interpolation Boolean polynomial p of b_Z^O.

$$i = 1$$
$$Z_1 = \emptyset, Z_0 = \{0\}, O_1 = \{0\}, O_0 = \{1\}$$
$$C = \{0\}$$
$$f = b_\emptyset^{\{0\}}, g = b_{\{0\}}^{\{1\}}, f + g = b_\emptyset^{\{0\}}$$
$$h_t = interpolate\_smallest\_lex(b_\emptyset^{\{0\}}) = 1$$

$$F = \emptyset, w = b_{\{0\}}^{\{1\}}$$

$$h_e = interpolate\_smallest\_lex(b_{\{0\}}^{\{1\}})$$

$$p = x_1 + h_e$$

We apply the Algorithm 4.4.1 again to find the interpolation polynomial q of $b_{\{0\}}^{\{1\}}$

$$i = 2$$

$$Z_1 = \emptyset, Z_0 = \{()\}, O_1 = \{()\}, O_0 = \emptyset$$

$$C = \{()\}$$

$$f = b_{\emptyset}^{\{()\}}, g = b_{\{()\}}^{\emptyset}, f + g = b_{\emptyset}^{\{()\}}$$

$$h_t = interpolate\_smallest\_lex(b_{\emptyset}^{\{()\}}) = 1$$

$$F = \emptyset, w = b_{\{()\}}$$

$$h_e = interpolate\_smallest\_lex(b_{\{()\}}) = 0$$

$$q = x_2$$

The result is $p = x_1 + x_2$.

Algorithm 4.4.2 computes the reduced lexicographical normal form of a polynomial w.r.t vanishing ideal of a variety without Gröbner basis computation. The algorithm calls the procedure $zeros(f, P)$ to find the set Z of all zeros of f in P, then the desired normal form is the smallest lexicographical interpolation polynomial of $b_Z^{P \backslash Z}$.

Algorithm 4.4.2 Reduced lexicographical normal form against variety
---
Input: Boolean polynomial f, P set of points in $\mathbb{Z}_2^n$.
Output: $nf\_by\_interpolate(f, P) = NF(f, I(P))$
  $Z = zeros(f, P)$
  Return $interpolate\_smallest\_lex(b_Z^{P \backslash Z})$

Example 4.4.5. We apply Algorithm 4.4.2 to find the normal form of $p = x_1 \cdot x_2 + x_1 + x_2$ with respect to the variety $P = \{(0,0), (0,1), (1,0)\}$. The set of zeros of p in P is $Z = \{(0,0)\}$. Therefore, the normal form of p with respect to P is

$$interpolate\_smallest\_lex(b_{\{(0,0)\}}^{\{(0,1),(1,0)\}}) = x_1 + x_2.$$

Given a monomial ordering $>$ and a set of points P. A monomial is called standard monomial of ideal I if it is not in the leading ideal of I. The set of standard monomials of $I(P)$ has the same cardinality as P. Based on this conclusion, Algorithm 4.4.3 is developed to compute all standard monomials of $I(P)$. The algorithm takes a random subset Z of P, then finds the smallest lex. interpolation polynomial p of $b_Z^{P \backslash Z}$. This polynomial is reduced w.r.t $I(P)$, hence its terms are not in the leading ideal of I. Therefore, all terms of p as well as their divisors are standard monomials of $I(P)$. The process continues until all $|P|$ standard monomials of $I(P)$ are found. In this algorithm, the procedure $random\_subset(P)$ will return a random subset of P, the $supp(p)$ is the support of P defined as the set of all terms (with non-zero coefficients) of p.

Algorithm 4.4.3 Standard monomials of I(P): standard_monomials_variety(P)

Input: P set of points in $\mathbb{Z}_2^n$.
Output: $S = \{t | \exists \text{ reduced } p \in I(P) : t \text{ term of } p\}$
  $S = \emptyset$
  while $|P| \neq |S|$ **do**
    $Z = random\_subset(P)$
    $p = interpolate\_smallest\_lex(b_Z^{P \setminus Z})$
    $S = S \cup supp(p)$
    $S = \{t \text{ term} \mid \exists s \in S : t \text{ divides } s\}$
  end while
  Return S

Example 4.4.6. Let $P = \{(0,0), (0,1), (1,0)\}$, we use Algorithm 4.4.3 to find all standard monomials of $I(P)$. Assume a random subset of P is $Z = \{(0,0)\}$, then

$$p = interpolate\_smallest\_lex(b_{\{(0,0)\}}^{\{(0,1),(1,0)\}}) = x_1 + x_2.$$

The support of p is $\{x_1, x_2\}$, hence $S = \{1, x_1, x_2\}$. At this point, S has 3 elements, so all the standard monomials of $I(P)$ are $1, x_1, x_2$.

After finding all standard monomials, we can obtain the leading monomials of the minimal Gröbner basis by collecting all minimal elements of the remaining Boolean monomials, see Algorithm 4.4.4

Algorithm 4.4.4 Leading monomials of a minimal Gröbner basis of I(P)

Input: P set of points in $\mathbb{Z}_2^n$.
Output: leading_monomials_variety$(P) = L(I(P))$
  $T = \{t \text{ Boolean term in } \mathbb{Z}_2[x_1, \ldots, x_n]\}$
  $R = T \setminus standard\_monomial\_variety(P)$
  Return minimal_elements(R)

Example 4.4.7. Consider again $P = \{(0,0), (0,1), (1,0)\}$. All Boolean terms in $\mathbb{Z}_2[x_1, \ldots, x_2]$ are $1, x_1, x_2$ and $x_1 x_2$. Using the result from Example 4.4.6, we get $R = \{x_1 x_2\}$ and the leading monomial of $I(P)$ is $x_1 x_2$.

The algorithm 4.4.5 constructs the reduced lexicographical Gröbner basis from the set of leading monomials.

Algorithm 4.4.5 $lex\_groebner\_basis\_points(P)$

Input: P a set of points in $\mathbb{Z}_2^n$.
  Return $\{t + nf\_by\_interpolate(t, P) | \ t \in L(I(P))\}$

Example 4.4.8. Continuing Example 4.4.7, the vanishing ideal of P has only one leading monomial, hence the reduced lexicographical Gröbner basis of $I(P)$ has only one polynomial. It is

$$x_1 x_2 + nf\_by\_interpolate(x_1 x_2, P) = x_1 x_2.$$

Table 4.3: Compare BDA and BMAlex

nvars	npoints	BDA	BMAlex		nvars	npoints	BDA	BMAlex
10	1000	0.1	2.9		100	100	0.7	0.3
20	1000	2.9	8		100	200	1.7	0.5
50	1000	11.1	8.6		100	500	10.8	2.1
100	1000	26	9		100	1000	29	7
200	1000	72	11		100	2000	86	34
500	1000	316	16		100	5000	446	637
1000	1000	929	24		100	10000	1549	9850

For a given set of points P, we presented two algorithms to compute the reduced Boolean Gröbner basis of vanishing ideal of P. To compare these two algorithms, we generate a random set of points P and also vary the number points and variables. From the Table 4.3, we can see that the algorithm from Brickenstein and Dreyer (BDA) is sensitive to the number of variables while BMAlex is sensitive to the number of points.

4.5 SAT and Interpolation approach

We propose a new method to compute the Gröbner basis of an elimination ideal of a given ideal. In this approach, we combine the power of all-solutions SAT Solver and interpolation. Firstly, we convert the ideal to a CNF formula F by the CNF encoder of Brickenstein. The benefit of this encoder is that no new variables are introduced. Then, we find all satisfying partial assignments V of F to important variables by All-SAT Solvers. Finally, we use interpolation to construct the Gröbner basis of the vanishing ideal of V.

Algorithm 4.5.1 SATElim

Input: An ideal I in $\mathbb{Z}[x_1, \ldots, x_n]$ containing $\langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle$, an integer $i \in [0, 1, \ldots, n]$
Output: reduced lexicographical Boolean Gröbner basis of ith elimination ideal of I
   P = all satisfying partial assignment d to $CNFEncoder(I)$
   /* d contains only variables in $\{x_{i+1}, \ldots, x_n\}$ */
   Return $lex\_groebner\_basis\_points(P)$

To prove the correctness of the Algorithm 4.5.1, we need the following theorem and its corollary proven in [BDG$^+$09].

Theorem 4.5.1. *Every ideal $I \subset \mathbb{Z}_2[x_1, \ldots, x_n]$ with $I \supset \langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle$ is radical.*

For such an ideal I in Theorem 4.5.1, the algebraic set $V(I)$ in $\mathbb{Z}_2^n$ and in the algebraic closure of $\mathbb{Z}_2^n$ are the same, hence we have

Corollary 4.5.2. *For an ideal $I \subset \mathbb{Z}_2[x_1, \ldots, x_n]$ with $I \supset \langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle$,*

$$I(V(I)) = I.$$

Now, we can prove the correctness of the Algorithm 4.5.1.

Theorem 4.5.3. *The algorithm 4.5.1 computes exactly the reduced lexicographical Boolean Gröbner basis of the ith elimination ideal of I.*

Proof. Algorithm 4.5.1 firstly computes P, the projection of all solutions of I on $\mathbb{Z}_2[Y]$, where $Y = \{x_{i+1}, \ldots, x_n\}$, in notation $P = \pi_Y(V(I))$.

The set $\pi_Y(V(I))$ is finite, so Zariski closed (since it is algebraic). Therefore, by the Closure Theorem,

$$\pi_Y(V(I)) = V(I \cap \mathbb{Z}_2[Y]). \tag{4.3}$$

This implies that the vanishing ideal of P is equal to $I(V(I \cap \mathbb{Z}_2[Y]))$. Applying Corollary 4.5.2 to $I \cap \mathbb{Z}_2[Y]$, we get

$$I(V(I \cap \mathbb{Z}_2[Y])) = I \cap \mathbb{Z}_2[Y]. \tag{4.4}$$

This proves that Algorithm 4.5.1 returns the reduced lexicographical Boolean Gröbner basis of $I \cap \mathbb{Z}_2[Y]$. $\square$

4.6 Experimental results

For the experiment, we use the All-SAT Solver source code from Yadgar [GSY04] and the Boolean interpolation source code from Brickenstein [BD13]. We compare the two approaches on the automata benchmarks. In the following, we will describe a simple automata benchmark. Let I be an ideal in the Boolean ring generated by

```
signal(4) + (1 + signal(1))(1 + state(2)),
signal(5) + (1 + signal(0)) state(1),
signal(6) + (1 + signal(2))(1 + signal(4)),
signal(7) + (1 + signal(3))(1 + signal(5)),
signal(8) + (1 + signal(4))(1 + signal(5)),
signal(9) + (1 + signal(7))(1 + signal(8)),
signal(10) + (1 + state(0)) signal(9),
signal(11) + signal(0)(1 + signal(10)),
next(0) + signal(11),
next(1) + signal(10),
next(2) + signal(6)
```

with variables $state(0)$, $state(1)$, $state(2)$, $signal(0), \ldots, signal(11)$ and $next(0)$, $next(1)$, $next(2)$.

To figure out relations between *states* variables and *next* variables, we have to eliminate all *signal* variables. This can be done by computing a Gröbner basis of an elimination ideal. The two approaches will be applied to solve this problem.

We limit time to 10 hours and memory to 50 GB. The sign $\times$ means error matrix-size exceed.

4.7 Conclusion

We have presented an alternative approach to compute Boolen Gröbner bases of Boolean elimination ideals. This approach is really efficient in case we want to eliminate most of the variables. In such a case, our approach can solve many hard instances that the classical approach can't. Our approach is very sensitive to the number of remaining variables. Time and memory can increase exponentially when this number increases.

Table 4.4: Comapre two approaches on some Automata benchmarks.

Problem informations			Time (in second)		Memory (in Megabyte)	
name	vars	Ivars	Gröbner basis	SATElim	Gröbner basis	SATElim
s27	18	6	0.07	0.02	84	83
s208	97	16	5.2	0.08	94	84
s298	140	28	×	21	-	185
s344	143	30	×	363	-	1251
s349	143	30	×	403	-	1267
s382	176	42	×	121	-	337
s386	192	12	25388	0.15	1394	84
s420	205	32	×	26	-	242
s444	201	42	×	158	-	334
s510	244	12	TO	0.16	-	84
s526	252	42	×	19510	-	16866
s641	219	38	×	32775	-	15310
s713	222	38	TO	34217	-	15370
s820	410	10	TO	0.20	-	85
s832	425	10	TO	0.17	-	85
s838	421	64	×	-	-	MO
s953	421	58	TO	1.13	-	104
s1196	528	36	TO	35116	-	20175
s1238	576	36	TO	15481	-	11544
s1423	626	148	TO	-	-	MO
s1488	699	12	TO	0.58	-	87
s1494	709	12	TO	0.58	-	87

Table 4.5: Comapre two approaches on some simplified benchmarks.

name	Time (in second)		Memory (in Megabyte)	
	Gröbner basis	SATElim	Gröbner basis	SATElim
H.S27	0.03	0.03	83	83
H.S208	0.03	0.07	84	83
H.S298	0.01	0.16	84	84
H.S344	0.12	7.32	85	154
H.S349	0.13	4.6	84	148
H.S382	17.08	2.78	217	124
H.S386	22.19	0.08	127	83
H.S420	4.94	0.12	92	84
H.S444	0.36	0.01	85	84
H.S510	4.4	0.09	109	84
H.S526	0.35	1.24	85	93
H.S641	5415	6.6	864	123
H.S713	3566	6.69	574	128
H.S820	4465	0.153	1331	85
H.S832	8215	0.16	1974	85
H.S838	47544	25	2842	253
H.S953	84	0.27	121	20.5

Chapter 5

Verification by abstraction and computer algebra techniques

This is a joint work with Oliver Marx from the department of Electrical and Computer Engineering, TU Kaiserslautern. Oliver Marx builded the abstract models on which I could build algebraic models and use computer algebra techniques to prove the desired properties.

5.1 Introduction

Verification is very important since it ensures that the design works correctly as expected. However, the usual methods (for example, using an instance of SAT solver, Binary Decision Diagram, simulation) can not verify efficiently the components that contain complicated datapaths, like multiplier and filter. Pavlenko at al [PWS+11] has proposed a computer algebra approach to solve this problem. They extract bit-level informations and transform them to polynomials in $\mathbb{Z}_{2^k}[X]/\langle x^2 - x : x \in X \rangle$. This means that coefficients of polynomials are finite length integers, but variables are still bit variables. The number of variables and polynomials will increase very much if the bitwidth is increasing. For instance, to model a signal of type 64 bits integer, we need 64 bit-variables.

Recent developments in abstraction techniques allow us to build abstract models at word level. For instances, we can reformulate the following bit operation $a << 1$, i.e. shift a 1 bit to right, to word operations $a \cdot 2$.

Our approach is as follows. We use abstraction techniques to lift bit operations to word operations whenever possible. This way, we can construct an abstract model. Then, we reformulate it as a system of polynomials in ring $\mathbb{Z}_{2^k}[x_1, \ldots, x_n]$. We try to order the variables such that the system has been already a Gröbner basis w.r.t lexicographical monomial ordering. Finally, the normal form is employed to prove the desired properties.

The description of the abstraction techniques is out of the scope of this thesis. It will appear in Oliver Marx's thesis. Here, we focus on how to build the algebraic model from the abstract design and prove its properties by computer algebra tools.

To evaluate our approach, we verify the global property of a multiplier and a FIR filter using the computer algebra system SINGULAR [DGPS12]. The result shows that our approach is much faster than the commercial verification tool from Onespin [One].

In this chapter, digital designs will be described in *Very High Speed Integrated Circuit Hardware Description Language* (VHDL) at *Register Transfer Level* (RTL).

Figure 5.1: Clock signal in form of a square wave
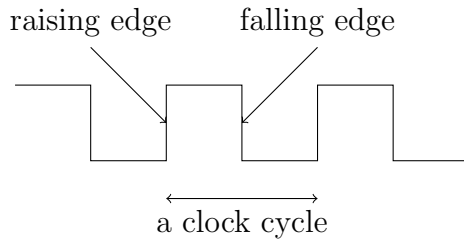
5.2 VHDL

VHDL is a standard language for describing the structure and function of digital electronic systems. It also allows us to verify the behavior of the design before translating it into real hardware.

In VHDL, a design consists of at least an entity which describes the interface and at least an architecture which describes the function. An entity is an object in VHDL containing information about input ports and output ports. An architecture contains processes, concurrent signal assignments, etc.

A simple signal assignments in VHDL is of the form

$$signal\_name <= expression$$

For example, $x <= a + b$.

A *process* is a sequence of statements. We may specify a list of signals as *sensitive list* of the process. In case the sensitive list is specified, the process is only activated when an event occurs (that is, the value of some signal in the sensitive list changes), the process suspends after the last statement. If more than one process is activated at same time, they are executed concurrently. The signal assignment in a process becomes effective only when the process suspends. Before that moment, they take their old values.

A concurrent signal assignments is the process which contains only one signal assignment statement. In this case, all signals on the left hand side of the assignment play role of the sensitive list.

To activate a process at a fixed frequency, a *clock signal* is included in the sensitive list of the process. A clock signal is a particular type of signal that oscillates between a high and a low state in the form of a square wave. The distance between two successive raising edges is a *clock cycle*, see Fig. 5.1. A statement whose execution depends on clock signal is called *synchronous statement*, the others are called *asynchronous statements*.

Example 5.2.1. Consider the following VHDL code

```
c<=a;
proc_ex: process (clk)
 begin
 if (raising_edge(clk)) then
    b<=a;
  end if;
end process proc_ex;
```

It consists of a process named *proc_ex*, and a concurrent signal assignment. The process is only sensitive to the clock signal. The signal assignment $b <= a$ is synchronous and is executed

only at raising edge of clock signal. In contrast, the concurrent signal assignment $c <= a$ is asynchronous and is executed whenever the value of a changes.

5.3 Algebraic models

We associate the synchronous assignment $a <= p(x, y, \dots)$ with $a_i - p(x_{i-1}, y_{i-1}, \dots)$, where p is a polynomials depending on the variable $x, y \dots$ and i is the considered state. The polynomial associated to the asynchronous assignment $a <= p(x, y, \dots)$ should be $a_i - p(x_i, y_i, \dots)$. The range of i depends on how many clock cycles we need to prove the desired properties.

Assignment	synchronous	asynchronous
$c <= a + b$	$c_i - a_{i-1} - b_{i-1}$	$c_i - a_i - b_i$
$c <= a * b$	$c_i - a_{i-1} \cdot b_{i-1}$	$c_i - a_i \cdot b_i$

When all signal assignments as well as the desired properties can be represented as polynomials over $\mathbb{Z}_{2^k}$, we can prove the property using Gröbner basis techniques and reduced normal forms. Let S be the set of all polynomials associated with signal assignments at all states involved in the proof goals, and I the ideal generated by S. Let p be the polynomial that represents the proof goal. Compute a Gröbner basis G of the ideal I. Then the property holds if and only if the normal form of p with respect to G is zero. Moreover, in most cases, we can order the variables with respect to the so-called *topological order*, such that S is a Gröbner basis w.r.t lexicographical monomial ordering.

The topological order of variables is computed as follows

1. Variables corresponding to the signal in the right hand side of a signal assignments are less than the one in the left hand side.

2. If two variables correspond to the same signal, but at different state, then the variable corresponding to the signal at higher state is greater.

With topological order of variables above and the lexicographical monomial ordering, the leading monomials of all polynomials in S are variables. If these variables are pairwise different, then S is a Gröbner basis of I.

5.4 Applications

We use our approach to verify the global property of a multiplier and a filter.

5.4.1 Multiplier

We consider a design of a multiplier which is used for multiplying two integers of the same bitwidth. The verification tool Onespin [One] needs 724 seconds to verify the global property of the original 24 bit multiplier design. Onespin also use symbolic method to verify the property. Some preprocessing techniques are applied at first, then bit-blasting is used to convert the problem to CNF and solved by SAT solvers. The paper [PWS$^+$11] shows that the CNF coming from bit-blasting of the design with complicated data path, like multiplier, is very hard for SAT solver.

To make the verification process faster, we first build an abstract design of this multiplier. Now, Onespin can do the job with the abstract design in 592 seconds. However, this cost is still not feasible since we want to verify the multiplier of higher bits. Therefore, we do a next step, building an algebraic model based on the abstract model, then using computer algebra techniques to verify it.

Consider a multiplier whose original design is written in the VHDL language and given in Appendix A.1. There are lots of type casting in this design. Moreover, most of the operations are at the bit level. To have a simple algebraic model, we need to build an abstract design of this multiplier. The new design will take 2 integers a and b as inputs and return $result$ as product of a and b. The other inputs (e.g clk, $reset$, and $start$) and output (e.g $ready$) are used to control the multiplying process.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.const.all;


ENTITY multiplier IS
    port (
        clk    : IN    std_logic;
        reset  : IN    std_logic;

        a      : IN    std_logic_vector(BIT_WIDTH-1 DOWNTO 0);
        b      : IN    std_logic_vector(BIT_WIDTH-1 DOWNTO 0);
        start  : IN    std_logic;

        ready  : OUT   std_logic;
        result : OUT   std_logic_vector(2*BIT_WIDTH-1 DOWNTO 0)
    );
END multiplier;


ARCHITECTURE behavior OF multiplier IS
    type state_type is (S_CSM_IDLE, S_CSM_CALCULATE);
    signal state: state_type;
    signal ready_r: std_logic;
    signal reg_a: unsigned(2*BIT_WIDTH-1 DOWNTO 0);
    signal reg_b: unsigned(BIT_WIDTH-1 DOWNTO 0);
    signal result_r: unsigned(2*BIT_WIDTH-1 DOWNTO 0);


BEGIN
    FSM: PROCESS(clk, reset)
    BEGIN
        if (reset='1') then
            state <= S_CSM_IDLE;
            ready_r <= '0';
        elsif (rising_edge(clk)) then
            if (state=S_CSM_IDLE) and (start='0') then
                state <= S_CSM_IDLE;
                ready_r <= '0';
            elsif (state=S_CSM_IDLE) and (start='1') then
                state <= S_CSM_CALCULATE;
                reg_a <= a;
                reg_b <= b;
                result_r <= 0;
                ready_r <= '0';
            elsif (state = S_CSM_CALCULATE) and (reg_b /= 0) then
```

```
            state <= S_CSM_CALCULATE;
            reg_a <= reg_a * 2;
            reg_b <= reg_b / 2;
            result_r <= result_r + reg_b(0) * reg_a;
            ready_r <= '0';
        elsif (state = S_CSM_CALCULATE) and (reg_b = 0) then
            state <= S_CSM_IDLE;
            reg_b <= 0;
            ready_r <= '1';
        end if;
    end if;
  END PROCESS FSM;
   ready <= ready_r;
   result <= result_r;
END behavior;
```

We want to prove the following property. At any state i, if $ready_i = 1$ then $r_i = a \cdot b$. However, $ready_i = 1$ if and only if $b_i = 0$, hence the property is equivalent to $r_i = a \cdot b$ whenever $b_i = 0$. This property is called the *global property* of the multiplier. To prove this property, we construct its algebraic model.

In this design, there are assignments whose right hand sides are not polynomials of word variables. The first one contains the integer division $b/2$, and the second one concerns extracting the least significant bit b_0 of the integer b, where

$$b = \sum_{j=0}^{k} 2^j b_j.$$

These two operations can be represented as polynomials over $\mathbb{Z}_{2^k}$ as follows

$$b_i' + 2 \cdot b_i - b_{i-1},$$

It contains a finite set of polynomials J describing the operations in the design and the polynomials describing the proof goals. The system J can be partitioned into an initial ideal $Init$, state ideals J_i, for i from 0 to bitwidth, where

$$
\begin{aligned}
Init &= \{a_0 - a, b_0 - b, r_0\}, \\
J_i &= \{a_i - 2 \cdot a_{i-1}, b_{i-1}' + 2 \cdot b_i - b_{i-1}, r_i - r_{i-1} - a_{i-1} \cdot b_{i-1}'\}.
\end{aligned}
$$

The state ideal J_i represents all operations performed at state i.

We can find a topological variable order such that J as well as $I = J \cup \{b_i\}$ is a Gröbner basis w.r.t lexicographical monomial ordering, e.g

$$r_k > \cdots > r_0 > a_k > \cdots > a_0 > a > b_k' > \cdots > b_0' > b_k > \cdots > b_0 > b.$$

With above ordering, the leading terms of all polynomials are variables, and these variables are pairwise different, hence the ideals I and J are Gröbner bases.

Polynomials for proof goal are $p_i = r_i - a \cdot b$. We compute the normal form of all p_i with respect to the ideal I. The desired property is satisfied if and only if all the normal forms are zero.

Example 5.4.1. As an illustration, we would verify the global property of a 2-bit multiplier. The proof goal is $r_i = a \cdot b$ whenever $b_i = 0$ for $0 \leq i \leq 2$. The proof goal involves 2 clock cycles. Therefore, the system J is the union of:

$$Init = \{a_0 - a, b_0 - b, r_0\},$$
$$J_1 = \{a_1 - 2 \cdot a_0, b'_0 + 2 \cdot b_1 - b_0, r_1 - r_0 - a_0 \cdot b'_0\}$$
$$J_2 = \{a_2 - 2 \cdot a_1, b'_1 + 2 \cdot b_2 - b_1, r_2 - r_1 - a_1 \cdot b'_1\}.$$

Polynomials for the proof goal are $p_1 := r_1 - a \cdot b$ and $p_2 := r_2 - a \cdot b$. We also add the polynomial b_i for the assumption $b_i = 0$ to J when checking p_i. The variables are ordered as follows:

$$r_2 > r_1 > r_0 > a_2 > a_1 > a_0 > a > b'_1 > b'_0 > b_2 > b_1 > b_0 > b.$$

With the lexicographical monomial ordering, the leading terms of all polynomials in $J \cup \{b_i\}$ are

$$a_0, b_0, r_0, a_1, b'_0, r_1, a_2, b'_1, r_2, \text{ and } b_i.$$

All of them are variables and pairwise different, hence $J \cup \{b_i\}$ are Gröbner bases for $i = 0, 1, 2$ by the product criterion.

Now, we compute the normal form of p_2 with respect to $I_2 := J \cup \{b_2\}$. The notation $f \xrightarrow{g} h$ means f is reduced to h by g.

$$p_2 := r_2 - a \cdot b \xrightarrow{r_2 - r_1 - a_1 \cdot b'_1} r_1 + a_1 \cdot b'_1 - a \cdot b$$

$$\xrightarrow{r_1 - r_0 - a_0 \cdot b'_0} r_0 + a_1 \cdot b'_1 + a_0 \cdot b'_0 - a \cdot b$$

$$\xrightarrow{r_0} a_1 \cdot b'_1 + a_0 \cdot b'_0 - a \cdot b$$

$$\xrightarrow{a_1 - 2 \cdot a_0} 2 \cdot a_0 \cdot b'_1 + a_0 \cdot b'_0 - a \cdot b$$

$$\xrightarrow{a_0 - a} 2 \cdot a \cdot b'_1 + a \cdot b'_0 - a \cdot b$$

$$\xrightarrow{b'_1 + 2 \cdot b_2 - b_1} a \cdot b'_0 - 4 \cdot a \cdot b_2 + 2 \cdot a \cdot b_1 - a \cdot b$$

$$\xrightarrow{b'_0 + 2 \cdot b_1 - b_0} -4 \cdot a \cdot b_2 + a \cdot b_0 - a \cdot b$$

$$\xrightarrow{b_2} a \cdot b_0 - a \cdot b$$

$$\xrightarrow{b_0 - b} 0$$

Similarly, we can prove that

$$NF(p_i \mid J \cup \{b_i\}) = 0, \text{ for all } i = 0, 1, 2.$$

Hence, the global property of this multiplier is satisfied.

The Table 5.1 presents the verification time (in seconds) of the global property of the multiplier with different bitwidths using the computer algebra system *Singular* [DGPS12]. The timing table shows that our approach can verify the global property of the multiplier of up to 1024 bits while Onespin is unable to verify it from 64 bit.

Table 5.1: Verify multiplier by abstraction and computer algebra

bit-width	nvars	time
32	133	0.05
64	261	0.26
128	517	1.83
256	1029	21
512	2053	283
1024	4101	6138

5.4.2 FIR Filter

FIR filters are special digital filters used in Digital Signal Processing (DSP) applications. The FIR stands for Finite Impulse Response. Original designs of FIR filters are not presented in this thesis, since it is too complicated. We consider at first a small FIR filter with its abstract design as follows:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity cf_fir_3_8_8 is
port(
    signal clock_c : in std_logic;
    signal reset : in signed(0 downto 0);
    signal data : in signed(7 downto 0);
    signal k0 : in signed(7 downto 0);
    signal k1 : in signed(7 downto 0);
    signal k2 : in signed(7 downto 0);
    signal k3 : in signed(7 downto 0);
    signal data_o : out signed(17 downto 0)
);
end entity cf_fir_3_8_8;

architecture rtl of cf_fir_3_8_8 is
    signal mult_0 : signed(15 downto 0);
    signal mult_1 : signed(15 downto 0);
    signal mult_2 : signed(15 downto 0);
    signal mult_3 : signed(15 downto 0);
    signal data_r_delayed_1 : signed(7 downto 0);
    signal data_r_delayed_2 : signed(7 downto 0);
    signal data_r_delayed_3 : signed(7 downto 0);
    signal data_r_delayed_4 : signed(7 downto 0);
    signal data_o_r_delayed_1 : signed(17 downto 0);
    signal data_o_r_delayed_2 : signed(17 downto 0);
    signal data_o_r_delayed_3 : signed(17 downto 0);
begin
    mult_0  <= k0 * data_r_delayed_1;
    mult_1  <= k1 * data_r_delayed_2;
    mult_2  <= k2 * data_r_delayed_3;
    mult_3  <= k3 * data_r_delayed_4;
    data_o <= data_o_r_delayed_3;
process(clock_c, reset) is
    begin
        if reset = 1 then
            data_r_delayed_1  <= 0;
```

```
        data_r_delayed_2   <= 0;
        data_r_delayed_3   <= 0;
        data_o_r_delayed_1 <= 0;
        data_o_r_delayed_2 <= 0;
        data_o_r_delayed_3 <= 0;
    elsif rising_edge(clock_c) then
        data_r_delayed_1   <= data;
        data_r_delayed_2   <= data_r_delayed_1;
        data_r_delayed_3   <= data_r_delayed_2;
        data_r_delayed_4   <= data_r_delayed_3;
        data_o_r_delayed_1 <= mult_0 + mult_1 + mult_2 + mult_3;
        data_o_r_delayed_2 <= data_o_r_delayed_1;
        data_o_r_delayed_3 <= data_o_r_delayed_2;
    end process;
end architecture rtl;
```

In this design, there are both synchronous assignments and asynchronous assignments. The names of signals are meanful, but they are too long. Therefore, we assign to each signal a new variable name. The Table 5.2 shows the signal names and their corresponding variable names.

Table 5.2: Signal names and their corresponding variable names

signal name	variable name
$data$	a
$data\_r\_delayed\_j$ $(j = 1, 2, 3, 4)$	b_j
kj $(j = 0, 1, 2, 3)$	k_j
$mult\_j$ $(j = 0, 1, 2, 3)$	m_j
$data\_o\_r\_delayed\_j$ $(j = 1, 2, 3)$	y_j
$data\_o$	z

All asynchronous and synchronous assignments together with their associated polynomials at state i are presented in Table 5.3 and Table 5.4.

Table 5.3: Transformations of all asynchronous assignments

VHDL assignments	Polynomials
$mult\_j <= k\_j * data\_r\_delayed\_(j + 1)$ with $j = 0, 1, 2, 3$	$m_{j,i} - k_{j,i} \cdot b_{j+1,i}$
$data\_o <= data\_o\_r\_delayed\_3$	$z_i - y_{3,i}$

Table 5.4: Transformations of all synchronous assignments

VHDL assignments	Polynomials
$data\_r\_delayed\_1 <= data$	$b_{1,i} - a_{i-1}$
$data\_r\_delayed\_(j + 1) <= data\_r\_delayed\_j$ with $j = 1, 2, 3$	$b_{j+1,i} - b_{j,i-1}$
$data\_o\_r\_delayed\_1 <= mult\_0 + mult\_1 + mult\_2 + mult\_3$	$y_{1,i} - m_{0,i-1} - m_{1,i-1} - m_{2,i-1} - m_{3,i-1}$
$data\_o\_r\_delayed\_(j + 1) <= data\_o\_r\_delayed\_j$ with $j = 1, 2$	$y_{j+1,i} - y_{j,i-1}$

We want to prove the following property of the design.

```
property property_h is
assume:
    during[t-4,t]: reset = 0;
prove:
```

```
    at  t :  mult_0  =  k0_i  *  prev ( data , 1 );
    at  t :  mult_1  =  k1_i  *  prev ( data , 2 );
    at  t :  mult_2  =  k2_i  *  prev ( data , 3 );
    at  t :  mult_3  =  k3_i  *  prev ( data , 4 );
    at  t :  data_o  =  prev (  mult_0  +  mult_1  +  mult_2  +  mult_3  , 3 );
end  property ;
```

We may understand the property above as follows: assume that there is no reset in 5 continue states (name these state $1, 2, 3, 4, 5$), prove that

- $mult\_0$ (at state 5) $= k0\_i$ (at state 5) $* data$ (at state 4),

- $mult\_1$ (at state 5) $= k1\_i$ (at state 5) $* data$ (at state 3),

- $mult\_2$ (at state 5) $= k2\_i$ (at state 5) $* data$ (at state 2),

- $mult\_3$ (at state 5) $= k3\_i$ (at state 5) $* data$ (at state 1),

- $data\_o$ (at state 5) $= mult\_0 + mult\_1 + mult\_2 + mult\_3$ (at state 2).

The polynomials for these proof goals are

- $p_0 := m_{0,5} - k_{0,5} \cdot a_4,$

- $p_1 := m_{1,5} - k_{1,5} \cdot a_3,$

- $p_2 := m_{2,5} - k_{2,5} \cdot a_2,$

- $p_3 := m_{3,5} - k_{3,5} \cdot a_1,$

- $p_4 := z_5 - m_{0,2} - m_{1,2} - m_{2,2} - m_{3,2}.$

Let G be the set of all polynomials in Table 5.3 and 5.4 at five states $1, 2, 3, 4, 5$. There are total 60 polynomials in G. The topological variable order of G at state i is

$$z_i > y_{3,i} > \cdots > y_{1,i} > m_{3,i} > \cdots > m_{0,i} > b_{3,i} > \cdots > b_{1,i} > a_i > k_{3,i} > k_{0,i}.$$

If some variables correspond to the same signals, but at different states, then the variable corresponding to the signal at higher state is larger. When the variables are ordered in this way, the leading terms (w.r.t lex) of all polynomials in G are variables and pairwise different. Hence, G is a Gröbner basis w.r.t lexicographical ordering. Therefore, to check the proof goals, we just need to check whether the normal form p_i w.r.t G is zero. The normal form of p_2 w.r.t G is computed as follows:

$$p_2 := m_{2,5} - k_{2,5} \cdot a_2 \xrightarrow{m_{2,5} - k_{2,5} \cdot b_{3,5}} k_{2,5} \cdot b_{3,5} - k_{2,5} \cdot a_2$$
$$\xrightarrow{b_{3,5} - b_{2,4}} k_{2,5} \cdot b_{2,4} - k_{2,5} \cdot a_2$$
$$\xrightarrow{b_{2,4} - b_{1,3}} k_{2,5} \cdot b_{1,3} - k_{2,5} \cdot a_2$$
$$\xrightarrow{b_{2,4} - a_2} 0$$

So the property corresponding to p_2 is satisfied. Similarly, the properties corresponding to $p_0, p_1,$ and p_3 are also satisfied. Now, we compute the normal form of p_4 w.r.t G.

$$p_4 := z_5 - m_{3,2} - m_{2,2} - m_{1,2} - m_{0,2} \xrightarrow{z_5 - y_{3,5}} y_{3,5} - m_{3,2} - m_{2,2} - m_{1,2} - m_{0,2}$$

$$\xrightarrow{y_{3,5}-y_{2,4}} y_{2,4} - m_{3,2} - m_{2,2} - m_{1,2} - m_{0,2}$$

$$\xrightarrow{y_{2,4}-y_{1,3}} y_{1,3} - m_{3,2} - m_{2,2} - m_{1,2} - m_{0,2}$$

$$\xrightarrow{y_{1,3}-m_{3,2}-m_{2,2}-m_{1,2}-m_{0,2}} 0$$

It implies that the property corresponding to p_4 is also satisfied.

To compare the timing of the verification by different tools, we consider the abstract design of a larger FIR filter presented in A.2. With this design, Singular just needs one second to verify the algebraic model, while Onespin needs 70 seconds to verify the same property with the abstract design, and would need approximately 600 years to verify that property with the original design.

5.5 Conclusion

Verification by abstraction and computer algebra techniques is a promising approach. We can verify a digital component containing a complicated data path much faster than the usual approach. By lifting bit operations to word operations, we can reduce the number of variables and polynomials by many order of magnitudes. As a result, the verification by algebraic method is more efficient with the help of abstraction techniques.

Appendix A

Designs of a multiplier and a filter

A.1 Multiplier

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.const.all;

ENTITY multiplier IS
   port (
      clk   : IN    std_logic;
      reset : IN    std_logic;

      a      : IN   std_logic_vector(BIT_WIDTH-1 DOWNTO 0);
      b      : IN   std_logic_vector(BIT_WIDTH-1 DOWNTO 0);
      start  : IN   std_logic;

      ready  : OUT std_logic;
      result : OUT std_logic_vector(2*BIT_WIDTH-1 DOWNTO 0)
      );
END multiplier;

ARCHITECTURE behavior OF multiplier IS
   type state_type is (IDLE, CALCULATING,OUTPUT);
   signal state: state_type;
   signal valid_r: std_logic;
   signal reg_a: unsigned(2*BIT_WIDTH-1 DOWNTO 0);
   signal reg_b: unsigned(BIT_WIDTH-1 DOWNTO 0);
   signal sum: unsigned(2*BIT_WIDTH-1 DOWNTO 0);
BEGIN
   FSM: PROCESS(clk,reset)
   BEGIN
      if (reset='1') then
         state <= IDLE;
         reg_a <= (others => '0');
         reg_b <= (others => '0');
         sum <= (others => '0');
         valid_r <= '0';
      elsif (rising_edge(clk)) then
         if (state=IDLE) and (start='0') then
            valid_r <= '0';
         elsif (state=IDLE) and (start='1') then
            state <= CALCULATING;
            reg_a <= resize(unsigned(a),2*BIT_WIDTH);
            reg_b <= unsigned(b);
            valid_r <= '0';
```

```vhdl
                    sum <= (others => '0');
            elsif (state=CALCULATING) then
                if (reg_b = 0) then
                    state <= IDLE;
                    valid_r <= '1';
                else
                    if reg_b(0) = '1' then
                        sum <= sum + reg_a;
                    end if;
                    reg_a(2*BIT_WIDTH-1 downto 0) <= reg_a(2*BIT_WIDTH-2
                        downto 0) & '0';
                    reg_b(BIT_WIDTH-1 downto 0) <= '0' & reg_b(BIT_WIDTH-1
                        downto 1);
                end if;
            end if;
        end if;
    END PROCESS FSM;

    ready <= valid_r;
    result <= std_logic_vector(sum);
END behavior;
```

```
property multiply is
dependencies:
    no_reset;
for timepoints:
    tb = t+1..BIT_WIDTH waits_for ready=1;
freeze:
    fa = a@t,
    fb = b@t;
assume:
    at t: CSM_IDLE;
    at t: start = 1;
prove:
    at tb: result = fa*fb;
end property;
```

A.2 FIR filter

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity cf_fir_33_16_16 is
port(
signal clock_c : in std_logic;
signal reset : in unsigned(0 downto 0);
signal data : in unsigned(15 downto 0);
signal k_0 : in unsigned(15 downto 0);
signal k_1 : in unsigned(15 downto 0);
signal k_2 : in unsigned(15 downto 0);
signal k_3 : in unsigned(15 downto 0);
signal k_4 : in unsigned(15 downto 0);
signal k_5 : in unsigned(15 downto 0);
signal k_6 : in unsigned(15 downto 0);
signal k_7 : in unsigned(15 downto 0);
signal k_8 : in unsigned(15 downto 0);
signal k_9 : in unsigned(15 downto 0);
```

```vhdl
signal k_10 : in unsigned(15 downto 0);
signal k_11 : in unsigned(15 downto 0);
signal k_12 : in unsigned(15 downto 0);
signal k_13 : in unsigned(15 downto 0);
signal k_14 : in unsigned(15 downto 0);
signal k_15 : in unsigned(15 downto 0);
signal k_16 : in unsigned(15 downto 0);
signal k_17 : in unsigned(15 downto 0);
signal k_18 : in unsigned(15 downto 0);
signal k_19 : in unsigned(15 downto 0);
signal k_20 : in unsigned(15 downto 0);
signal k_21 : in unsigned(15 downto 0);
signal k_22 : in unsigned(15 downto 0);
signal k_23 : in unsigned(15 downto 0);
signal k_24 : in unsigned(15 downto 0);
signal k_25 : in unsigned(15 downto 0);
signal k_26 : in unsigned(15 downto 0);
signal k_27 : in unsigned(15 downto 0);
signal k_28 : in unsigned(15 downto 0);
signal k_29 : in unsigned(15 downto 0);
signal k_30 : in unsigned(15 downto 0);
signal k_31 : in unsigned(15 downto 0);
signal k_32 : in unsigned(15 downto 0);
signal k_33 : in unsigned(15 downto 0);
signal data_o : out unsigned(37 downto 0));
end entity cf_fir_33_16_16;

architecture rtl of cf_fir_33_16_16 is
    signal reg_0 : unsigned(31 downto 0);
    signal reg_1 : unsigned(31 downto 0);
    signal reg_2 : unsigned(31 downto 0);
    signal reg_3 : unsigned(31 downto 0);
    signal reg_4 : unsigned(31 downto 0);
    signal reg_5 : unsigned(31 downto 0);
    signal reg_6 : unsigned(31 downto 0);
    signal reg_7 : unsigned(31 downto 0);
    signal reg_8 : unsigned(31 downto 0);
    signal reg_9 : unsigned(31 downto 0);
    signal reg_10 : unsigned(31 downto 0);
    signal reg_11 : unsigned(31 downto 0);
    signal reg_12 : unsigned(31 downto 0);
    signal reg_13 : unsigned(31 downto 0);
    signal reg_14 : unsigned(31 downto 0);
    signal reg_15 : unsigned(31 downto 0);
    signal reg_16 : unsigned(31 downto 0);
    signal reg_17 : unsigned(31 downto 0);
    signal reg_18 : unsigned(31 downto 0);
    signal reg_19 : unsigned(31 downto 0);
    signal reg_20 : unsigned(31 downto 0);
    signal reg_21 : unsigned(31 downto 0);
    signal reg_22 : unsigned(31 downto 0);
    signal reg_23 : unsigned(31 downto 0);
    signal reg_24 : unsigned(31 downto 0);
    signal reg_25 : unsigned(31 downto 0);
    signal reg_26 : unsigned(31 downto 0);
    signal reg_27 : unsigned(31 downto 0);
    signal reg_28 : unsigned(31 downto 0);
    signal reg_29 : unsigned(31 downto 0);
```

```vhdl
    signal reg_30 : unsigned(31 downto 0);
    signal reg_31 : unsigned(31 downto 0);
    signal reg_32 : unsigned(31 downto 0);
    signal reg_33 : unsigned(31 downto 0);
    signal data_r_delayed_1 : unsigned(15 downto 0);
    signal data_r_delayed_2 : unsigned(15 downto 0);
    signal data_r_delayed_3 : unsigned(15 downto 0);
    signal data_r_delayed_4 : unsigned(15 downto 0);
    signal data_r_delayed_5 : unsigned(15 downto 0);
    signal data_r_delayed_6 : unsigned(15 downto 0);
    signal data_r_delayed_7 : unsigned(15 downto 0);
    signal data_r_delayed_8 : unsigned(15 downto 0);
    signal data_r_delayed_9 : unsigned(15 downto 0);
    signal data_r_delayed_10 : unsigned(15 downto 0);
    signal data_r_delayed_11 : unsigned(15 downto 0);
    signal data_r_delayed_12 : unsigned(15 downto 0);
    signal data_r_delayed_13 : unsigned(15 downto 0);
    signal data_r_delayed_14 : unsigned(15 downto 0);
    signal data_r_delayed_15 : unsigned(15 downto 0);
    signal data_r_delayed_16 : unsigned(15 downto 0);
    signal data_r_delayed_17 : unsigned(15 downto 0);
    signal data_r_delayed_18 : unsigned(15 downto 0);
    signal data_r_delayed_19 : unsigned(15 downto 0);
    signal data_r_delayed_20 : unsigned(15 downto 0);
    signal data_r_delayed_21 : unsigned(15 downto 0);
    signal data_r_delayed_22 : unsigned(15 downto 0);
    signal data_r_delayed_23 : unsigned(15 downto 0);
    signal data_r_delayed_24 : unsigned(15 downto 0);
    signal data_r_delayed_25 : unsigned(15 downto 0);
    signal data_r_delayed_26 : unsigned(15 downto 0);
    signal data_r_delayed_27 : unsigned(15 downto 0);
    signal data_r_delayed_28 : unsigned(15 downto 0);
    signal data_r_delayed_29 : unsigned(15 downto 0);
    signal data_r_delayed_30 : unsigned(15 downto 0);
    signal data_r_delayed_15 : unsigned(15 downto 0);
    signal data_r_delayed_32 : unsigned(15 downto 0);
    signal data_r_delayed_33 : unsigned(15 downto 0);
    signal data_r_delayed_34 : unsigned(15 downto 0);
    signal data_o_r_delayed_1 : unsigned(31 downto 0);;
    signal data_o_r_delayed_2 : unsigned(31 downto 0);;
    signal data_o_r_delayed_3 : unsigned(31 downto 0);;
    signal data_o_r_delayed_4 : unsigned(31 downto 0);;
    signal data_o_r_delayed_5 : unsigned(31 downto 0);;
    signal data_o_r_delayed_6 : unsigned(31 downto 0);;
    signal data_o_r_delayed_7 : unsigned(31 downto 0);;
    signal data_o_r_delayed_8 : unsigned(31 downto 0);;
    signal data_o_r_delayed_9 : unsigned(31 downto 0);;
begin
    reg_0  <= k_0 * data_r_delayed_1;
    reg_1  <= k_1 * data_r_delayed_2;
    reg_2  <= k_2 * data_r_delayed_3;
    reg_3  <= k_3 * data_r_delayed_4;
    reg_4  <= k_4 * data_r_delayed_5;
    reg_5  <= k_5 * data_r_delayed_6;
    reg_6  <= k_6 * data_r_delayed_7;
    reg_7  <= k_7 * data_r_delayed_8;
    reg_8  <= k_8 * data_r_delayed_9;
    reg_9  <= k_9 * data_r_delayed_10;
```

```vhdl
        reg_10 <= k_10 * data_r_delayed_11;
        reg_11 <= k_11 * data_r_delayed_12;
        reg_12 <= k_12 * data_r_delayed_13;
        reg_13 <= k_13 * data_r_delayed_14;
        reg_14 <= k_14 * data_r_delayed_15;
        reg_15 <= k_15 * data_r_delayed_16;
        reg_16 <= k_16 * data_r_delayed_17;
        reg_17 <= k_17 * data_r_delayed_18;
        reg_18 <= k_18 * data_r_delayed_19;
        reg_19 <= k_19 * data_r_delayed_20;
        reg_20 <= k_20 * data_r_delayed_21;
        reg_21 <= k_21 * data_r_delayed_22;
        reg_22 <= k_22 * data_r_delayed_23;
        reg_23 <= k_23 * data_r_delayed_24;
        reg_24 <= k_24 * data_r_delayed_25;
        reg_25 <= k_25 * data_r_delayed_26;
        reg_26 <= k_26 * data_r_delayed_27;
        reg_27 <= k_27 * data_r_delayed_28;
        reg_28 <= k_28 * data_r_delayed_29;
        reg_29 <= k_29 * data_r_delayed_30;
        reg_30 <= k_30 * data_r_delayed_31;
        reg_31 <= k_31 * data_r_delayed_32;
        reg_32 <= k_32 * data_r_delayed_33;
        reg_33 <= k_33 * data_r_delayed_34;
        data_o <= data_o_r_delayed_9;
process(clock_c,reset) is
    begin
        if reset = 1 then
            data_r_delayed_1  <= 0;
            data_r_delayed_2  <= 0;
            data_r_delayed_3  <= 0;
            data_r_delayed_4  <= 0;
            data_r_delayed_5  <= 0;
            data_r_delayed_6  <= 0;
            data_r_delayed_7  <= 0;
            data_r_delayed_8  <= 0;
            data_r_delayed_9  <= 0;
            data_r_delayed_10 <= 0;
            data_r_delayed_11 <= 0;
            data_r_delayed_12 <= 0;
            data_r_delayed_13 <= 0;
            data_r_delayed_14 <= 0;
            data_r_delayed_15 <= 0;
            data_r_delayed_16 <= 0;
            data_r_delayed_17 <= 0;
            data_r_delayed_18 <= 0;
            data_r_delayed_19 <= 0;
            data_r_delayed_20 <= 0;
            data_r_delayed_21 <= 0;
            data_r_delayed_22 <= 0;
            data_r_delayed_23 <= 0;
            data_r_delayed_24 <= 0;
            data_r_delayed_25 <= 0;
            data_r_delayed_26 <= 0;
            data_r_delayed_27 <= 0;
            data_r_delayed_28 <= 0;
            data_r_delayed_29 <= 0;
            data_r_delayed_30 <= 0;
```

```vhdl
            data_r_delayed_31 <= 0;
            data_r_delayed_32 <= 0;
            data_r_delayed_33 <= 0;
            data_r_delayed_34 <= 0;
            data_o_r_delayed_1 <= 0;
            data_o_r_delayed_2 <= 0;
            data_o_r_delayed_3 <= 0;
            data_o_r_delayed_4 <= 0;
            data_o_r_delayed_5 <= 0;
            data_o_r_delayed_6 <= 0;
            data_o_r_delayed_7 <= 0;
            data_o_r_delayed_8 <= 0;
            data_o_r_delayed_9 <= 0;
        elsif rising_edge(clock_c) then
            data_r_delayed_1  <= data;
            data_r_delayed_2  <= data_r_delayed_1;
            data_r_delayed_3  <= data_r_delayed_2;
            data_r_delayed_4  <= data_r_delayed_3;
            data_r_delayed_5  <= data_r_delayed_4;
            data_r_delayed_6  <= data_r_delayed_5;
            data_r_delayed_7  <= data_r_delayed_6;
            data_r_delayed_8  <= data_r_delayed_7;
            data_r_delayed_9  <= data_r_delayed_8;
            data_r_delayed_10 <= data_r_delayed_9;
            data_r_delayed_11 <= data_r_delayed_10;
            data_r_delayed_12 <= data_r_delayed_11;
            data_r_delayed_13 <= data_r_delayed_12;
            data_r_delayed_14 <= data_r_delayed_13;
            data_r_delayed_15 <= data_r_delayed_14;
            data_r_delayed_16 <= data_r_delayed_15;
            data_r_delayed_17 <= data_r_delayed_16;
            data_r_delayed_18 <= data_r_delayed_17;
            data_r_delayed_19 <= data_r_delayed_18;
            data_r_delayed_20 <= data_r_delayed_19;
            data_r_delayed_21 <= data_r_delayed_20;
            data_r_delayed_22 <= data_r_delayed_21;
            data_r_delayed_23 <= data_r_delayed_22;
            data_r_delayed_24 <= data_r_delayed_23;
            data_r_delayed_25 <= data_r_delayed_24;
            data_r_delayed_26 <= data_r_delayed_25;
            data_r_delayed_27 <= data_r_delayed_26;
            data_r_delayed_28 <= data_r_delayed_27;
            data_r_delayed_29 <= data_r_delayed_28;
            data_r_delayed_30 <= data_r_delayed_29;
            data_r_delayed_31 <= data_r_delayed_30;
            data_r_delayed_32 <= data_r_delayed_31;
            data_r_delayed_33 <= data_r_delayed_32;
            data_r_delayed_34 <= data_r_delayed_33;
            data_o_r_delayed_1 <= reg_0 + reg_1 + reg_2 + reg_3
            + reg_4 + reg_5 + reg_6 + reg_7 + reg_8 + reg_9
            + reg_10 + reg_11 + reg_12 + reg_13 + reg_14 + reg_15
            + reg_16 + reg_17 + reg_18 + reg_19 + reg_20 + reg_21
            + reg_22 + reg_23 + reg_24 + reg_25 + reg_26 + reg_27
            + reg_28 + reg_29 + reg_30 + reg_31 + reg_32 + reg_33;
            data_o_r_delayed_2 <= data_o_r_delayed_1;
            data_o_r_delayed_3 <= data_o_r_delayed_2;
            data_o_r_delayed_4 <= data_o_r_delayed_3;
            data_o_r_delayed_5 <= data_o_r_delayed_4;
```

```vhdl
                data_o_r_delayed_6 <= data_o_r_delayed_5;
                data_o_r_delayed_7 <= data_o_r_delayed_6;
                data_o_r_delayed_8 <= data_o_r_delayed_7;
                data_o_r_delayed_9 <= data_o_r_delayed_8;
            end if;
        end process;
end architecture rtl;
```

Appendix B

Implementation source codes

B.1 Codes used in Chapter 3

```cpp
#include <simp/SimpSolver.h>
#include "utils/System.h"
namespace Minisat {}

using namespace Minisat;

class GBhelpSolver: public SimpSolver {
public:
    // Constructor: makes en empty solver
    GBhelpSolver();

    // Destructor: removed this from memory
    virtual ~GBhelpSolver() {}
    void selectGBinput(std::vector<Clause*>& GBinput);
    bool GBinput_ok(const std::vector<Clause*>& GBinput);
    void computeGB(const std::vector<Clause*>& GBinput, std::vector<std::::
        vector<int> >& GBoutput);
    void selectGBoutput(std::vector<std::vector<int> > GBoutput, std:::
        vector<std::vector<int> >& selectedGBoutput);
    void GBlearning();
    void decide_activateGB();
    void decide_deactivateGB(const std::vector<Clause*>& GBinput, const
        std::vector<std::vector<int> >& selectedGBoutput);
    void addGBlearnt();
    void setGBparams();


    // BGB: Constant for Boolean Groebner bases
    int         gbhelp;                 // Options for Groebner basis learning
        schemes (0=none, 1=Kuechlin, 2 = density base)
    double      density_limit;      // Density of Groebner bases input.
    double      gb_factor_inc;
    int         gb_max;                 // Maximum size of GB learnt clauses
    bool        only_confl;             // only conflict clauses are selected
        from GB output
    bool        aggressive;             // looking for locally new binary lit
        aggressively
    bool        LBD2;                   // collet also GB learnt clauses with
        LBD = 2
    int         gb_factor;
    long int    gb_counter;         // For Kuechlin approach
    bool        gb_on;                  // Activate Groebner bases help
    std::vector<std::vector<int> >    gb_learnts;
```

```cpp
    double       total_gb_time;
    uint64_t     gbs, gbadds, gbzeros, maxdepth; // GB statistics
    std::set<int> all_levels;        // to store all decision level of a
        GB_learnt clause

    virtual void extended_clause_learning() {
      GBlearning();
    }
    virtual void add_clauses_from_extended_learning() { addGBlearnt(); }
    virtual void prepare_extended_learning() { setGBparams(); }

};

#include "GBhelpSolver.h"
#include "embed.h"

// =====================

static const char* _gb = "GROEBNER BASES";
static IntOption    opt_gbhelp              (_gb, "gbhelp",        "
   Controls learning by Groebner basis (0=none, 1=Kuechlin, 2 = density
   base)", 2, IntRange(0,2));
static IntOption    opt_gb_factor_inc    (_gb, "gb_factor_inc",      "
    Control GB frequency ", 2 , IntRange(1,1000));
static DoubleOption opt_density_limit    (_gb, "density_limit",     "
   Upper bound of density of GB Input", 1.4, DoubleRange(1,false,100,
   false));
static IntOption    opt_gb_max              (_gb, "gb_max",       " the
   maxiamum size of GB learnt clauses", 5 , IntRange(1,50));
static BoolOption   opt_only_confl       (_gb, "only_confl",  " Collect
    only conflict clauses from GB output", false);
static BoolOption   opt_aggressive       (_gb, "aggressive",  " Collect
    only conflict clauses from GB output", false);
// For LBD
static BoolOption   opt_LBD2              (_gb, "LBD2",       " Collect
    also  clauses from GB output with  LBD = 2", false);



// =====================

GBhelpSolver::GBhelpSolver(): SimpSolver()    // For Boolean Groebner
   bases
  , gbhelp             (opt_gbhelp)
  , density_limit      (opt_density_limit)
  , gb_factor_inc      (opt_gb_factor_inc)
  , gb_max             (opt_gb_max)
  , only_confl         (opt_only_confl)
  , aggressive         (opt_aggressive)
  , LBD2               (opt_LBD2)
  , gb_factor          (1)
  , gb_counter         (0)
  , gb_on              (true)
  , total_gb_time      (0)
  , gbs(0), gbadds(0), gbzeros(0) { }


// =====================
```

```cpp
void GBhelpSolver::selectGBinput(std::vector<Clause*>& GBinput)
{
    GBinput.clear();

    bool select_on = false;

    if (gbhelp == 1) {
        select_on = true;
    }
    if ((gb_on) and (gbhelp == 2)){
        select_on =true;
    }
    for (int i = 0; i < confl_concern.size(); i++){
        Clause& c = *(confl_concern[i]);
        // BGB select input
        if (select_on){
                if (gbhelp == 1){
                        if (c.size() <= 8) GBinput.push_back(
                            confl_concern[i]);
                }
                if (gbhelp == 2){
                        Clause* cl = confl_concern[i];
                        if ( c.size() <=4 ) GBinput.push_back(cl);
                }
                if (GBinput.size() > 6) select_on =false;
        }
        if (!select_on) break;
    }
}

// =====================

bool GBhelpSolver::GBinput_ok(const std::vector<Clause*>& GBinput)
{
        if (GBinput.size() < 4) return false;
        if ((gbhelp==1) and (GBinput.size() > 6)) return false;

//        if (gb_on and (gbhelp == 2)){
        if (gbhelp == 2){
        // check density of GB input
                std::set<int> input_varset;

                for (unsigned int i = 0; i < GBinput.size(); i++){
                        Clause& c = *GBinput[i];
                        for (int j = 0; j < c.size(); j++) input_varset.
                            insert(var(c[j]));
                }
                float density;
                density = (float) input_varset.size()/GBinput.size();
                if ( density > density_limit) return false;
        }
        return true;
}

// =====================
```

```cpp
void GBhelpSolver::computeGB(const std::vector<Clause*>& GBinput, std::::
    vector<std::vector<int> >& GBoutput)
{
        // BGB : Input is ok, compute GB with command "groebner_basis" in
            Polybori

        BEGIN_PBORI_EMBED();
        exec("ll = list()");
        for (unsigned int i = 0; i < GBinput.size(); i++){
                Clause& c = *GBinput[i];
                std::vector<int> cla;
                cla.clear();
                for (int j = 0; j < c.size(); j++){
                        int v = 2*var(c[j])+(int) sign(c[j]);
                        cla.push_back(v);
                }
                Interpreter::globals()["l"] = cla;
                Interpreter::globals()["max_size"] = gb_max;
                exec("ll.append(l)");
        }
        exec("from GBhelp import *");
        exec("res = gbLearning(ll,max_size)");
        //
        int nsize = eval("len(res)");
//      int new_lit_in_binary_counter = 0;
        std::vector<std::vector<int> > allclauses(nsize);
        for (int idx = 0; idx < nsize; ++idx) {
            allclauses[idx] = eval("lambda idx: res[idx]")(idx);
        }
        GBoutput = allclauses;

        END_PBORI_EMBED();
}

// =====================

void GBhelpSolver::selectGBoutput(std::vector<std::vector<int> > GBoutput
    , std::vector<std::vector<int> >& selectedGBoutput)
{
        for (unsigned int idx = 0; idx < GBoutput.size(); ++idx) {
                std::vector<int>&  cl = GBoutput[idx];
                bool good_cl = true;
                if (good_cl) selectedGBoutput.push_back(cl);
        }
}


// =====================
void GBhelpSolver::decide_activateGB()
{
        if (gb_counter % gb_factor == 0) gb_on = true;
}


void GBhelpSolver::decide_deactivateGB(const std::vector<Clause*>&
    GBinput, const std::vector<std::vector<int> >& selectedGBoutput)
{
        if (!aggressive or (gbs>100)){
```

```cpp
                gb_on = false;
                return;
        }
}
// =====================

void GBhelpSolver::GBlearning()
{
        if (gbhelp==0) return;
        if (gbhelp==2) gb_counter++;
        // Activate GB if in case
        if (!gb_on) decide_activateGB();

        if ((!gb_on) and (gbhelp==2)) return;
        // Select GB input
        std::vector<Clause*> GBinput;
        GBinput.clear();
        double select_time = 0;
        double init_select_time = cpuTime();
        selectGBinput(GBinput);
        select_time = cpuTime() - init_select_time;

        double gb_time = 0;
        double init_gb_time = cpuTime();
        // Check GB input
        if (!GBinput_ok(GBinput)) {
                return;
        }
        // count satisfied GB input for KZ
        if ((gbhelp==1)) {
                gb_counter++;
                if (!gb_on) return;
        }
        // Compute Reduced Boolean Groebner basis
        std::vector<std::vector<int> > GBoutput;
        GBoutput.clear();
        computeGB(GBinput, GBoutput);

        // Select good clauses from GB output
        std::vector<std::vector<int> > selectedGBoutput;
        selectedGBoutput.clear();
        selectGBoutput(GBoutput, selectedGBoutput);

        // Store selected GB learnt clauses
        for (unsigned int i = 0; i < selectedGBoutput.size(); i++){
                gb_learnts.push_back(selectedGBoutput[i]);
        }

        // Consider to deactivate GB basis
        decide_deactivateGB(GBinput, selectedGBoutput);

        // For GB Statistics
        gbs++;
        if (selectedGBoutput.size() == 0) gbzeros++;
        else gbadds += selectedGBoutput.size();
        gb_time = cpuTime() - init_gb_time;
        total_gb_time = total_gb_time + gb_time + select_time;
}
```

```cpp
// =====================
// Add GB learnt clauses
//

void GBhelpSolver::addGBlearnt()
{
    if (gbhelp == 0) return;
    // For ND_GB, gb_counter is the number of conflicts in a search
    // so we must reset it when solver is restarted

    if (gbhelp == 2) gb_counter = 0;

    if (gb_factor < 1e6 ) gb_factor = (int) gb_factor*gb_factor_inc;

    for (unsigned int i = 0; i< gb_learnts.size(); i++) {
        vec<Lit> lits;
        lits.clear();
        for (unsigned int j = 0; j < gb_learnts[i].size(); j++) {
            int k = gb_learnts[i][j];
            Lit lit = toLit(k);
            lits.push(lit);
        }
        addClause(lits);
    }
    gb_learnts.clear();
}

// =====================

void GBhelpSolver::setGBparams()
{
    if (gbhelp==0) return;
    // Set parameters for KZ
    if (gbhelp == 1) {
        gb_factor       = 1;
        gb_factor_inc   = 2;
        gb_max          = 2;
        only_confl      = false;
        aggressive      = false;
        LBD2            = false;
    }
}
```

```python
from polybori.cnf import *
from polybori.blocks import declare_ring, Block
from polybori.gbcore import groebner_basis
from polybori.memusage import *
#from polybori.PyPolyBoRi import *
def list_var_index(poly):
        l = list(poly.vars_as_monomial().variables())
        res = list()
        for var in l:
                res.append(var.index())
        return res

def tolits(poly,var_map):
        # convert polynomial clauses to CNF
```

```
        lv = list_var_index(poly)
        lits = list()
        polyset = poly.set()
        for i in lv:
                if polyset.subset0(i).empty():
                        lits.append(2*var_map[i])
                else:
                        lits.append(2*var_map[i]+1)
        return lits

def gbLearning(ll,max_size=2):
        """
        Compute the groebner basis of corr. Polynomials
        of clauses in ll and collect all new binary clauses
        """
        ll.sort(key = len)
        var_map = list()
        ideal = list()
        for l in ll:
                for lit in l:
                        var = lit >> 1
                        if var not in var_map:
                                var_map.append(var)
        ring = declare_ring([Block("x",len(var_map))])
        enc = CNFEncoder(ring)
        for l in ll:
                poly = ring.one()
                for lit in l:
                        ind = var_map.index(lit >> 1)
                        if lit&1 :
                                poly = poly*(ring.variable(ind)+1)
                        else:
                                poly = poly*ring.variable(ind)
                ideal.append(poly)
        gb = groebner_basis(ideal,other_ordering_first=False)

        res = list()
        for poly in gb:
                if (poly.n_variables() <= max_size) and (poly not in
                    ideal):
                        polys = enc.polynomial_clauses(poly)
                        for p in polys:
                                lits = tolits(p,var_map)
                                res.append(lits)
        return res
```

B.2 Codes used in Chapter 4

```
# Optimize Buchberger Moeller Algorithm
# for Boolean polynomial and lex ordering
#------------------------
# -*- coding: utf-8 -*-
# Author: Hung
# Run in Sage, a free a mathematics software
#------------------------

import resource
```

```python
import sys
import signal

from sage.all import *
from polybori.gbrefs import load_file
from time import *
from polybori.cnf import *
from polybori.gbcore import groebner_basis
from polybori.interpolate import *
# For use process
from subprocess import Popen, PIPE
from polybori import Polynomial, Variable
from Pla2Boole import *
#----------------------------------
import random
#from polybori.blocks import declare_ring
from polybori.blocks import Block, declare_ring
from polybori.specialsets import monomial_from_indices

#---------------------------
def time_expired(n, stack):
    print 'EXPIRED :', ctime()

def encode_points_as_comp_list(P):
    n = len(P[0])
    m = len(P)
    Q = [[int(P[i][j]) for i in xrange(m)] for j in xrange(n)]
    return Q

def update_irre(irre,L,u,pointer):
    j= pointer
    while j<len(L):
        us=set(u)
        if us.issubset(L[j]):
            irre.add(j)
        j+=1
    return irre

def list2bset(l,one,zero):
    new=one
    for i in xrange(len(l)):
        new=new.change(l[i])
    return new

def updateB(B,mono,one,zero):
    new=list2bset(mono,one,zero)
    B.append(new)
    return

def updateB2(B,mono,one,zero):
    B.append(mono)
    return

def updateG(G,B,mono,one,zero,N_last_row,m,nrows):
    N2=N_last_row[m:m+nrows]
    f = list2bset(mono,one,zero)
    for j in reversed(N2.nonzero_positions()):
        f=f.union(B[j])
```

```
        G.append(Polynomial(f))
        return G

def updateG2(G,B,mono,one,zero,N_last_row,m,nrows):
    N2=N_last_row[m:m+nrows]
    f = list()
    f.append(mono)
    for j in reversed(N2.nonzero_positions()):
            f.append(B[j])
    G.append(f)
    return G

#------------------------------
def BM(P):
    m = len(P)
    G=[]
    if len(P)==0:
        G = [1]
    else:
        nvars = len(P[0])
        ring = declare_ring([Block("x",nvars)])
        zero=ring.zero().set()
        one=ring.one().set()
        global x
        x = ring.variable

        Q=encode_points_as_comp_list(P)
        del P

        L=[list()]
        addL=[]
        irre=set()
        B=[one]
        ptr= 0
        current_var_idx=nvars-1
        #Matrix initialize
        x_ks=matrix(GF(2),Q[current_var_idx])
        I= matrix.identity(GF(2),m)
        row=[1 for i in xrange(m)]
        M=matrix(GF(2),row)
        M_for_rank= M.augment(I[0:1,:])

        ext=True
        nrows=1
        while current_var_idx!=-1:
            if ptr not in irre:
                base_mono=list(L[ptr])
                current_mono = base_mono+[current_var_idx]

                M_row = M[ptr:ptr+1,:]
                new_row= x_ks.elementwise_product(M_row)
                v = matrix(GF(2),new_row)

                if nrows < m:
                    n_i= I[nrows:nrows+1,:]
                    v_ex = v.augment(n_i)
                else:
                    if ext:
```

```python
                        n_i= matrix(GF(2),1,m)
                        ext= False
                v_ex = v.augment(n_i)
                N=M_for_rank.stack(v_ex)
                N.echelonize()
                N_last_row= N[-1]
                N1= N_last_row[:m]
                if N1.is_zero():
                    irre= update_irre(irre,L,base_mono,ptr)
                    G=updateG2(G,B,current_mono,one,zero,N_last_row,m,nrows)
                else:
                    nrows+=1
                    M=M.stack(v)
                    M_for_rank= N
                    addL.append(current_mono)
                    updateB2(B,current_mono,one,zero)
            ptr+=1
            if ptr>= len(L):
                L=L+addL
                current_var_idx-=1
                x_ks=matrix(GF(2),Q[current_var_idx])
                ptr=0
                addL=[]
                irre=set()
        return G
#----------------------------------------
def lex_gb_noes(Sols,nvars,r):
        offset = 0
        S = bset_from_strings(r,Sols,offset)
        var_as_mono = Monomial(r)
        for i in xrange(offset, r.n_variables()):
                var_as_mono *=r.variable(i)
        res = None
        res = lex_groebner_basis_points(S,var_as_mono)
        return res
#----------------------------------------

def random_example(nvars,npoints):
    def get():
        line = ''
        for idx in xrange(nvars):
            line += random.choice('01')
        return line
    return [get() for elt in xrange(npoints)]

#----------------------------------------
def main():
    resource.setrlimit(resource.RLIMIT_CPU, (36000,36001))
    signal.signal(signal.SIGXCPU, time_expired)
    print 'Starting:', ctime()
    npoints= 1000
    #nvars=100
    print "npoints = ", npoints
    #print "nvars = ", nvars
    print "nvars    Brickenstein    BM"
    #print "npoints    Brickenstein    BM"
    nV=[10,20,50,100,200,500,1000]
    nV=[10000]
```

```python
    #nP=[100,200,500,1000,2000,5000,10000]

    for n in nV:
        nvars= n
        #npoints=n
        P=random_example(nvars,npoints)

        #print "******By Brickenstein********"
        #t   = time()
        #import polybori
        #r = polybori.Ring(nvars)
        #sys.setrecursionlimit(10000)
        #G1=lex_gb_noes(P,nvars,r)
        #t1=time()-t

        #print "******By Buchberger_M********"
        t   = time()
        G2=BM(P)
        t2= time()-t
        print nvars,"    ",round(t2,1)
        #print npoints,nvars,"    ",round(t1,1),"    ",round(t2,1)
    return

if __name__== "__main__":
    main()

# -*- coding: utf-8 -*-
# Author: Hung
#-----------------------
# Find relation between states and next states
# using all-solution SAT and interpolation
#-----------------------

import resource
import sys
import signal

from polybori.gbrefs import load_file
from time import *
from polybori.cnf import *
from polybori.gbcore import groebner_basis
from polybori.interpolate import *
# For use process
from subprocess import Popen, PIPE
from polybori import Polynomial, Variable
from Pla2Boole import *
#------------------------------------

################################
# Load test-files
################################
def get_data(number,k):
        data = load_file('/u/n/nguyen/polybori-work/sat_interpolation/
            test_file/S'+str(number)+'.py')
        if k > len(data.initial_states):
          raise ValueError, "k must less than or equal to the number of
            states"
```

71

```python
            data.ideal = data.ideal+data.initial_states[:k]
            return data
            #nvars = 2*data.num_states + data.num_signals


#################################
# Convert ideal to CNF
#################################
def toCNF(ideal):
        enc = CNFEncoder(ideal[0].ring())
        res = enc.dimacs_cnf(ideal)
        res += '\n'
        return res


#################################
# Extract solution info from
# Avi_all_sat output
#################################
def get_redPLA(avi_output):
        res =list()
        counter = 0
        for line in avi_output.split('\n'):
                if line.startswith('0') or line.startswith('1'):
                        res.append(line)
                        counter+=1
        #print 'number of SAT solutions = ', counter
        return res


#################################
# Solve all-solution SAT problem
#################################
def Avi_all_sat(CNFs,nImportant_vars):
        # length of a batch
        M = str(100)
        N = str(nImportant_vars)
        process = Popen(["batch_all_sat","-",N,M],bufsize = 4096, stdin=
            PIPE, stdout=PIPE)
        all_sat_output = process.communicate(CNFs)[0]
        res = get_redPLA(all_sat_output)
        return res


#################################
# Interpolation
#################################
def lex_gb_noes(Sols,nvars,r):
        offset = r.n_variables() - nvars
        S = bset_from_strings(r,Sols,offset)
        var_as_mono = Monomial(r)
        for i in xrange(offset, r.n_variables()):
                var_as_mono *=r.variable(i)
        res = None
        res = lex_groebner_basis_points(S,var_as_mono)
        return res


#################################
# Compute Groebner basis of
# elimination ideal
# by all-solution SAT
```

```python
# and interpolation
################################
def SATElim(data):
        t      = time()
        ideal = data.ideal
        ring = data.ring
        N = 2*data.num_states
        CNFs = toCNF(ideal)
        Sols = Avi_all_sat(CNFs,N)
        gb = lex_gb_noes(Sols,N,ring)
        return gb


################################
# Compute Groebner basis
################################
def GBElim(data):
    ideal = data.ideal
    ring  = data.ring

    ring2        = ring.clone(ordering=block_dp_asc, blocks=[data.
        num_signals])
    ideal2       = [ring2(p) for p in ideal]

    gb  = groebner_basis(ideal2)
    res = [poly for poly in gb if poly.navigation().value() >= data.
        num_signals ]
    return res


################################
# Test all benchmarks
################################

def time_expired(n, stack):
    print 'EXPIRED :', ctime()

def main():
    d={27:3, 208:8, 298:14, 344:15, 349:15, 382:21, 386:6, 420:16, 444:21,
        510:6, 526:21, 641:19, 713:19, 820:5, 832:5, 838:32, 953:29,
        1196:18, 1238:18, 1423:74, 1488:6, 1494:6}
    resource.setrlimit(resource.RLIMIT_CPU, (36000,36001))
    signal.signal(signal.SIGXCPU, time_expired)
    print 'Starting:', ctime()

    #Parse input
    method=sys.argv[1]
    num  = int(sys.argv[2])
    inum = 0

    print ""
    print "S"+ str(num)
    if method == "g":
        print "GB"
    elif method == "s":
        print "SAT"

    t  = time()
    #Get data
    data = get_data(num,inum)
```

```python
    if method == "g":
        gb = GBElim(data)
    elif method == "s":
        gb = SATElim(data)
    #Resource statistics
    GBtime = time()-t
    GBmem  = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss / 1000

    #Output to screen
    print "time  = ", GBtime
    print "mem   = ", GBmem, "M"
    print 'End     :', ctime()
    exit()
    return

if __name__== "__main__":
    main()
```

B.3 Codes used in Chapter 5

```bash
#!/bin/bash
#===============================================
# Check global property of an abstract multiplier
# design using computer algebra techniques
#===============================================
filename="multiplier.txt"

python << END
#-----------------------------------------------
bw=$1
fn = "$filename"
#-----------------------------------------------
# Help methods
#-----------------------------------------------

# indexing
def E(*args):
    res=args[0]
    for i in xrange(1,len(args)):
        res = res + "_" + str(args[i])
    return res

def mk_vars(name,bw):
    return [E(name,i) for i in reversed(xrange(bw+1))]

# Expand b to bit variables
def expand(b,bw):
    res = b
    for i in xrange(bw):
        res = res +" -" +str(2**i) +"*" +E(b,0,i)
    return res

# make new ring
def mk_ring(bw,all_vars,ordering,name):
    line= "ring "+ name+ " = (integer,2,"+ str(bw)+"),("
    for var in all_vars:
        line += var + ","
```

```python
        line= line[:-1]+")," + ordering
        lines.append(line)

# set options
def set_options(*args):
    for i in xrange(len(args)):
        line = "option("+ args[i]+ ")"
        lines.append(line)

# define an ideal
def mk_ideal(poly_list,start_index,name):
    res="ideal "+name+"= ("
    for poly in poly_list:
        p = "poly p"+ str(start_index)+ "=" + poly
        lines.append(p)
        res = res + "p"+str(start_index)+","
        start_index+=1
    res= res[:-1] + ")"
    lines.append(res)
    return start_index

# generate all step ideal
def mk_step_ideals(polys_block):
    start_index=0
    for i in xrange(len(polys_block)):
        name = "ide"+str(i)
        start_index=mk_ideal(polys_block[i],start_index,name)

# Make global ideal
def mk_global_ideal(polys_block):
    name = "ide"
    all_polys=[]
    for polys in polys_block:
        all_polys+=polys
    mk_ideal(all_polys,0,name)

# reduce ideal at step k+1 w.r.t ideal at step k for k = 0..bw
def successive_reduce(bw):
    for i in xrange(bw):
        line= "ide"+str(i+1)+"= reduce(ide"+str(i+1)+",ide"+str(i)+")"
        lines.append(line)

# check the proofgoal
def check_proofgoal(bw):
    lines.append("poly proofgoal = 0")
    lines.append("poly pready = 0")
    lines.append("ideal ideready = 0")
    for i in xrange(bw):
        p1= "proofgoal = "+E("r",i+1)+ "- a*b"
        lines.append(p1)
        p2= "pready = "+E("b",i+1)
        lines.append(p2)
        p3= "ideready = ide+pready"
        lines.append(p3)
        p5="reduce(proofgoal,ideready)"
        lines.append(p5)
    lines.append("quit")
```

```python
# write to a text file
def write2file(lines):
    new = open(fn, "w")
    for line in lines:
        new.write(line)
        new.write(";\n")
    new.close()
#----------------------------------------
# ring data
#----------------------------------------
r_vars     = mk_vars("r",bw)
ready_vars = mk_vars("ready",bw)
a_vars     = mk_vars("a",bw)
b_vars     = mk_vars("b",bw)
bz_vars      = mk_vars("bz",bw)
#all_vars   =r_vars + ready_vars + a_vars + ["a","b"] + bz_vars + b_vars
all_vars   =r_vars + ready_vars + a_vars + ["a"] + bz_vars + b_vars+["b"]
ordering   = "lp"


#----------------------------------------
# Poly data
#----------------------------------------
polys_block=[]
polys=[]
polys.append("r_0 - 0")
polys.append("a_0 - a")
polys.append("b_0 - b")
polys_block.append(polys)

for i in xrange(bw):
    polys=[]
    # Equation for r
    poly1= E("r",i+1) +E("-r",i)   +E("-a",i)   +E("*bz",i)
    polys.append(poly1)
    # Equation for a
    poly2= E("a",i+1) + E("-2*a",i)
    polys.append(poly2)
    # Equations model operator shift b left.
    poly3= E("2*b",i+1)+ E("-b",i) + E("+bz",i)
    polys.append(poly3)
    polys_block.append(polys)

#----------------------------------------
# Main
#----------------------------------------
lines=[]

mk_ring(bw,all_vars,ordering,"r")
set_options("notWarnSB")
mk_global_ideal(polys_block)
check_proofgoal(bw)

write2file(lines)
#----------------------------------------
END

time Singular $filename
```

```bash
#!/bin/bash
#================================================
# Verify the global property of an filter
# using computer algebra techniques
#================================================
filename="filter2.txt"

textfile=$1

python << END
#------------------------------------------------
fn = "$filename"
nc=42 # number of cycles
#------------------------------------------------
# Help methods
#------------------------------------------------

# indexing
def E(*args):
    res=args[0]
    for i in xrange(1,len(args)):
        res = res + "_" + str(args[i])
    return res

# generate a list of variables name_0,...,name_n
def mk_vars(name,n):
    return [E(name,i) for i in reversed(xrange(n+1))]

# Expand b to bit variables
def expand(b,bw):
    res = b
    for i in xrange(bw):
        res = res +" -" +str(2**i) +"*" +E(b,0,i)
    return res

# remove whitespace from a formula
def clean(s):
    res=""
    for i in xrange(len(s)):
        if s[i]!=" ":
            res+=s[i]
    return res

def add_index(v,k,res):
    if v.isdigit():
        res += v
    else:
        res += E(v,k)
    return res

def resolve_bitvar(s):
    res=s
    if s.find("[")== -1:
        return res
    res = res.replace("[","_")
    res = res.replace("]","")
    return res
```

77

```python
def one_side_eq(s):
    res = s
    res=res.replace("+","++")
    res=res.replace("-","+")
    res=res.replace("++","-")
    res=res.replace("=","-")
    return res

def parse_shift_right(s,k,bw):
    res=[]
    ss=s.split(">>")
    var=ss[0]
    ks = int(ss[1])
    for i in xrange(bw-ks):
        l = E(var,k+1,i) + "-" +E(var,k,i+ks)
        res.append(l)
    for i in xrange(bw-ks,bw):
        l = E(var,k+1,i)
        res.append(l)
    return res

def parse_eq(s,k):
    s=clean(s)
    s=s.replace("<=","=")
    s=resolve_bitvar(s)
    if s.find("==") >=0:
        s=s.replace("==","=")
        instant = True
    else:
        instant = False
    eq_index= s.find("=")
    res=""
    v=""
    i=0
    while i < len(s):
        if s[i] not in ["=","*","+","-"]:
            v+=s[i]
            if i == (len(s)-1):
                if instant:
                    res = add_index(v,k+1,res)
                else:
                    res = add_index(v,k,res)
        else:
            if (i<=eq_index) or instant:
                res = add_index(v,k+1,res)
            else:
                res = add_index(v,k,res)
            res += s[i]
            v=""
        i+=1
    res = one_side_eq(res)
    return res

def parse(textfile,ival):
    file = open(textfile, "r")
    eqs= file.read()
    file.close()
    res=[]
```

```python
    for i in xrange(ival):
        res.append([])
    for s in eqs.split("\n"):
        s=s.replace(";","")
        if s.find("=") >=0:
            for i in xrange(ival):
                tmp= parse_eq(s,i)
                res[i].append(tmp)
        elif s.find(">>") >=0:
            for i in xrange(ival):
                tmp= parse_shift_right(s,i,bw)
                res[i]+= tmp
    return res

#---------------------------------------------

# make new ring
def mk_ring(bw,all_vars,ordering,name):
    line= "ring "+ name+ " = (integer,2,"+ str(bw)+"),("
    for var in all_vars:
        line += var + ","
    line= line[:-1]+"),"+ ordering
    lines.append(line)

# set options
def set_options(*args):
    for i in xrange(len(args)):
        line = "option("+ args[i]+ ")"
        lines.append(line)

# define an ideal
def mk_ideal(poly_list,start_index,name):
    res="ideal "+name+"= ("
    for poly in poly_list:
        p = "poly p"+ str(start_index)+ "=" + poly
        lines.append(p)
        res = res + "p"+str(start_index)+","
        start_index+=1
    res= res[:-1] + ")"
    lines.append(res)
    return start_index

def mk_global_ideal(polys_block):
    name = "ide"
    all_polys=[]
    for polys in polys_block:
        all_polys+=polys
    mk_ideal(all_polys,0,name)

# check the proofgoal
def check_proofgoal():
    lines.append("poly proofgoal = data_o_42 - (data_i_34*k0_i_35 +
        data_i_33*k1_i_35 + data_i_32*k2_i_35 + data_i_31*k3_i_35 +
        data_i_30*k4_i_35 + data_i_29*k5_i_35 + data_i_28*k6_i_35 +
        data_i_27*k7_i_35 + data_i_26*k8_i_35 + data_i_25*k9_i_35 +
        data_i_24*k10_i_35 + data_i_23*k11_i_35 + data_i_22*k12_i_35 +
        data_i_21*k13_i_35 + data_i_20*k14_i_35 + data_i_19*k15_i_35 +
```

```python
                data_i_18*k16_i_35 + data_i_17*k17_i_35 + data_i_16*k18_i_35 +
                data_i_15*k19_i_35 + data_i_14*k20_i_35 + data_i_13*k21_i_35 +
                data_i_12*k22_i_35 + data_i_11*k23_i_35 + data_i_10*k24_i_35 +
                data_i_9*k25_i_35 + data_i_8*k26_i_35 + data_i_7*k27_i_35 +
                data_i_6*k28_i_35 + data_i_5*k29_i_35 + data_i_4*k30_i_35 +
                data_i_3*k31_i_35 + data_i_2*k32_i_35 + data_i_1*k33_i_35)")
    lines.append("reduce(proofgoal,ide)")
    lines.append("quit")


# write to a text file
def write2file(lines):
    new = open(fn, "w")
    for line in lines:
        new.write(line)
        new.write(";\n")
    new.close()
#----------------------------------------
# Generate list of variables
data_o_vars            = mk_vars("data_o",nc)

data_o_r_delayed_vars=[]
for i in reversed(xrange(7)):
    data_o_r_delayed_vars += mk_vars("data_o_r_delayed_"+str(i+1),nc)

reg_vars=[]
for i in reversed(xrange(34)):
    reg_vars += mk_vars("reg"+str(i),nc)

data_i_r_delayed_vars=[]
for i in reversed(xrange(34)):
    data_i_r_delayed_vars += mk_vars("data_i_r_delayed_"+str(i+1),nc)

data_i_vars= mk_vars("data_i",nc)

k_vars=[]
for i in reversed(xrange(34)):
    k_vars += mk_vars("k"+str(i)+"_i",nc)
# variable ordering and monomial ordering
all_vars   = data_o_vars + data_o_r_delayed_vars + reg_vars +
    data_i_r_delayed_vars + data_i_vars + k_vars
ordering   = "lp"
# Poly data
polys_block= parse("$textfile",nc)
# Main
lines=[]
mk_ring(38,all_vars,ordering,"r")
set_options("notWarnSB")
mk_global_ideal(polys_block)
check_proofgoal()
# write to file
write2file(lines)
#------------------------------------------------------------
END

time Singular $filename
```

Bibliography

[ABKR00]   J. Abbott, A. Bigatti, M. Kreuzer, and L. Robbiano. Computing ideals of points. *Journal of Symbolic Computation*, 30(4):341–356, 2000.

[AS09]   Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[BD09]   Michael Brickenstein and Alexander Dreyer. PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials. *Journal of Symbolic Computation*, 44(9):1326–1345, 2009. Effective Methods in Algebraic Geometry.

[BD13]   Michael Brickenstein and Alexander Dreyer. Gröbner-free normal forms for boolean polynomials. *Journal of Symbolic Computation*, 48(0):37–53, 2013.

[BDG$^+$09]   Michael Brickenstein, Alexander Dreyer, Gert-Martin Greuel, Markus Wedler, and Oliver Wienand. New developments in the theory of gröbner bases and applications to formal verification. *Journal of Pure and Applied Algebra*, 213(8):1612–1635, 2009. Theoretical Effectivity and Practical Effectivity of Gröbner Bases.

[BHvMW09]   A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[BMS00]   Luís Baptista and João Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability, 2000.

[Bri10]   Michael Brickenstein. *Boolean Gröbner bases – Theory, Algorithms and Applications*. PhD thesis, University of Kaiserslautern, Germany, 2010.

[Buc85]   B. Buchberger. Gröbner bases: an algorithmic method in polynomial ideal theory. In N. K. Bose, editor, *Multidimensional Systems Theory*, pages 184–232. D. Reidel Publishing Company, 1985.

[CEI96]   M. Clegg, J. Edmonds, and R. Impagliazzo. Using the Gröbner basis algorithm to find proofs of unsatisfiability. *Proceedings of the Twenty-eighth Annual ACM Symposium on the Theory of Computing*, pages 174–183, 1996.

[CK07]   Christopher Condrat and Priyank Kalla. A Gröbner basis approach to CNF-formulae preprocessing. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 618–631. Springer, 2007.

[DGPS12]   Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann. SINGULAR 3-1-6 — A computer algebra system for polynomial computations. `http://www.singular.uni-kl.de`, 2012.

[DLL62]   Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[DP60]   Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[EB05]   Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the 8th international conference on Theory and Applications of Satisfiability Testing*, SAT'05, pages 61–75, Berlin, Heidelberg, 2005. Springer-Verlag.

[ES04]   Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 333–336. Springer Berlin / Heidelberg, 2004.

[GP02]   G.-M. Greuel and G. Pfister. *A SINGULAR Introduction to Commutative Algebra*. Springer Verlag, 2002.

[GP07]   Gert-Martin Greuel and Gerhard Pfister. *A Singular Introduction to Commutative Algebra*. Springer Publishing Company, Incorporated, 2nd edition, 2007.

[GSK98]   Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the 15th national conference on Artificial intelligence*, pages 431–437. American Association for Artificial Intelligence, 1998.

[GSY04]   Orna Grumberg, Assaf Schuster, and Avi Yadgar. Memory efficient all-solutions sat solver and its application for reachability analysis. In *In Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD*, pages 275–289. Springer, 2004.

[JS06]   Winfried Just and Brandilyn Stigler. Computing gröbner bases of ideals of few points in high dimensions. *ACM Commun. Comput. Algebra*, 40(3-4):67–78, September 2006.

[LS04]   Reinhard Laubenbacher and Brandilyn Stigler. A computational algebra approach to the reverse engineering of gene regulatory networks. *Journal of Theoretical Biology*, 229(4):523–537, 2004.

[Lun08]   Samuel Lundqvist. Mathematical methods in computer science. chapter Complexity of Comparing Monomials and Two Improvements of the Buchberger-Möller Algorithm, pages 105–125. Springer-Verlag, Berlin, Heidelberg, 2008.

[MB82]   H. Michael Möller and Bruno Buchberger. The construction of multivariate polynomials with preassigned zeros. In *Proceedings of the European Computer Algebra Conference on Computer Algebra*, EUROCAM '82, pages 24–31, London, UK, UK, 1982. Springer-Verlag.

[MSSSS96]   Joao P. Marques-Silva, Joo P. Marques Silva, Karem A. Sakallah, and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *in Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, 1996.

[One]   Onespin Solutions GmbH, Germany onespin 360mv. `www.onespin-solutions.com`.

[PWS+11]   E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz, A. Dreyer, F. Seelisch, and G. Greuel. Stable: A new qf-bv smt solver for hard verification problems combining boolean reasoning with computer algebra. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[RD06]   G. Van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., Bristol, United Kingdom, November 2006.

[SNC09]   Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending sat solvers to cryptographic problems. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 244–257, Berlin, Heidelberg, 2009. Springer-Verlag.

[zGG03]   Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, second edition, 2003.

[ZK10]   Christoph Zengler and Wolfgang Küchlin. Extending clause learning of SAT solvers with Boolean Gröbner bases. In *Proceedings of the 12th international conference on Computer algebra in scientific computing*, CASC'10, pages 293–302, Berlin, Heidelberg, 2010. Springer-Verlag.

Wissenschaftlicher Werdegang

1990 – 2002	Besuch der Schule in Vinh Long, Vietnam.
2002	Abschluss an der Pham Hung High School, Vinh Long, Vietnam.
2006	Bachelorabschluss am Fachbereich Mathematik an der Can Tho Universität, Can Tho, Vietnam. **Bachelorarbeit:** *Solving linear programming problem by affine scaling method* **Supervisor:** M.sc. Ho Huu Hoa
2010	Masterabschluss am Fachbereich Mathematik an der TU Kaiserslautern. **Masterarbeit:** *Modular Algorithms for Computing a Generating Set of the Syzygy Module* **Supervisor:** Prof. Dr. Gerhard Pfister
seit 04/2012	Promotionstudent am Fachbereich Mathematik an der TU Kaiserslautern. **Dissertation:** *Combinations of Boolean Gröbner Bases and SAT Solvers* **Supervisor:** Prof. Dr. Gerhard Pfister

Curriculum Vitae

1990 – 2002	Elementary, secondary, and high school in Vinh Long, Vietnam.
2002	Graduation from Pham Hung High School, Vinh Long, Vietnam.
2006	Bachelor's degree in mathematics from Can Tho University, Vietnam. **Thesis:** *Solving linear programming problem by affine scaling method* **Supervisor:** M. sc. Ho Huu Hoa
2010	Master's degree in mathematics from TU Kaiserslautern **Thesis:** *Modular Algorithms for Computing a Generating Set of the Syzygy Module* **Supervisor:** Prof. Dr. Gerhard Pfister
from 04/2012	Ph.D studies in mathematics at TU Kaiserslautern. **Dissertation:** *Combinations of Boolean Gröbner Bases and SAT Solvers* **Supervisor:** Prof. Dr. Gerhard Pfister

Declaration

I hereby declare that this thesis is my own work and effort, and that no other sources than those listed have been used.

The third chapter of this thesis has been publisched in the Proceeding of 2nd Young Researcher Symposium 2013 by Fraunhofer Verlag, ISBN 978-3-8396-0628-5.

I am not in a second examination process right now.

Kaiserslautern, October 2014

Thanh Hung Nguyen