Verification using Satisfiability Checking, Predicate Abstraction, and Craig Interpolation

Himanshu Jain

September 2008

CMU-CS-08-146

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research is sponsored by the Gigascale Systems Research Center (GSRC), the Semiconductor Research Corporation (SRC), the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) and the Army Research Office (ARO).

Keywords: Formal methods, model checking, abstraction, refinement, bounded model checking, Boolean satisfiability, non-clausal SAT solvers, DPLL, general matings, unsatisfiable core, craig interpolation, proofs of unsatisfiability, linear diophantine equations, linear modular equations (linear congruences), linear diophantine disequations

To my parents.

Abstract

Automatic verification of hardware and software implementations is crucial for building reliable computer systems. Most verification tools rely on decision procedures to check the satisfiability of various formulas that are generated during the verification process. This thesis develops new techniques for building efficient decision procedures and adds new capabilities to the existing decision procedures for certain logics.

Boolean satisfiability (SAT) solvers are used heavily in verification tools as decision procedures for propositional logic. Most state-of-the-art SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and require the input formula to be in Conjunctive Normal Form (CNF). However, typical formulas that arise in practice are non-clausal, that is, not in CNF. Converting a general formula to CNF introduces overhead in the form of new variables and may destroy the structure of the initial formula, which can be useful to check satisfiability efficiently. We present two non-clausal SAT algorithms that operate on the Negation Normal Form (NNF) of the given formula. The NNF of a formula is usually more succinct than the CNF of the formula. The first algorithm is based on the idea of General Matings developed by Andrews in 1981. We develop techniques for performing search space pruning, learning, non-chronological backtracking in the context of a General Matings based SAT solver. The second algorithm applies the DPLL algorithm to NNF formulas. We devise new algorithms for performing Boolean Constraint Propagation (BCP), a key task in the DPLL algorithm.

Most hardware verification tools convert a high level design into a low level representation called a netlist for verification. However, algorithms that operate at the netlist level are unable to exploit the structure of the higher abstraction levels such as register transfer level, and thus, are less scalable. This thesis proposes the use of predicate abstraction for verifying register transfer level (RTL) Verilog. Predicate abstraction is a technique introduced for software verification. There are two challenges when applying predicate abstraction to circuits: (i) The computation of the abstract model in the presence of a large number of predicates, and (ii) discovery of suitable word-level predicates for abstraction refinement. We address the first problem using a technique called predicate clustering. We address the second problem by computing weakest pre-conditions of Verilog statements in order to obtain new word-level predicates during abstraction refinement.

An alternative technique for finding new predicates for refinement is based on the computation of Craig interpolants. Efficient algorithms are known for computing interpolants in rational and real linear arithmetic. We focus on subsets of integer linear arithmetic. Our main results are polynomial time algorithms for obtaining proofs of unsatisfiability and interpolants for conjunctions of linear diophantine equations, linear modular equations (linear congruences), and linear diophantine disequations. We show the utility of our interpolation algorithms for discovering modular/divisibility predicates in a counterexample guided abstraction refinement (CEGAR) framework. This has enabled verification of simple programs that cannot be checked using existing CEGAR based model checkers.

Acknowledgments

I was extremely lucky to have Ed Clarke as my thesis advisor. Ed's enthusiasm and encouragement kept me going. He spent countless hours listening to my half-baked ideas and helped me refine them into this thesis. He also taught me the importance of choosing the right research problems and focusing on both theory and practice in research. My deepest gratitude to Ed for all the things he taught me. He is truly an exemplary advisor. Ed's better half, Martha, made me feel at home away from home. I have fond memories of the numerous excellent parties that Martha organized for Ed's graduate students. Thank you Martha.

I was very fortunate to have an outstanding thesis committee consisting of Randy Bryant, Aarti Gupta, Ken McMillan, and Frank Pfenning. The comments and insights from them have helped improve both the results and the presentation of this thesis. A major part of this thesis is based on the *horizontal/vertical path forms* idea that I first learned in a Math Logic class taught by Peter Andrews. My special thanks to Peter for his meticulously taught course and for pointing me to the relevant literature on horizontal/vertical path forms.

I have benefited greatly from my collaborators Daniel Kroening, Orna Grumberg, Constantinos Bartzis, and Natasha Sharygina. My thanks to Daniel, Orna, Constantinos, and Natasha for their advice, comments, and insightful ideas. I am also grateful to the NEC verification group for two very fruitful summer internships. My undergraduate advisor Supratik Chakraborty first introduced me to the area of formal verification and got me started with the research in this area. My thanks to Supratik.

My big thanks to Denny Marous, Deb Cavlovich, Catherine Copetas, Sharon Burks, and Linda Gentile for making life at CMU so much easier. Thanks to Ryan and Sue Ann for tolerating me as their officemate. I am very grateful to Pankaj, Nishant, Muralidhar, Flavio, Sagar, Anubhav for their advice on both research and non-research related matters. I would also like to thank my other friends including Pradeep, Amitabh, Sandeep, Varun(s), Deepak, Pranjal for making my stay at CMU enjoyable. I am sure I am forgetting many other people who made this journey possible, my apologies and thanks to them.

Finally, I would like to thank my mother Pratibha, my father Dhanraj, my sisters Heena and Pranjal, and my brother Prateek for their love and encouragement.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Computer systems form an integral part of our day to day life. They are increasingly being used in safety critical applications such as automobiles, medical devices, aircrafts, and nuclear power plants. Automatic verification of the underlying hardware and software is crucial for building reliable computer systems. The goal of this thesis is to develop techniques and tools for obtaining scalable verification tools.

A *decision procedure* for a logic is an algorithm that reports whether a formula given in that logic is satisfiable or unsatisfiable. Decision procedures act as the reasoning engines in modern verification tools. In the first part of this thesis we focus on an important logic, namely the Boolean (propositional) logic. Our contributions and related work are described in Section 1.1. Most hardware verification tools convert a high-level design into a gate-level representation called *netlist* for verification. However, algorithms that operate at the netlist level are unable to exploit the structure of the higher abstraction levels, and can be less scalable. We develop techniques for verifying hardware designs at a higher level of abstraction than the netlist level. We describe our contributions and related work in Section 1.2. Recent hardware and software verification techniques expect the decision procedures to also provide proofs of unsatisfiability and Craig interpolants. We present our results on computing proofs of unsatisfiability and interpolants in Section 1.3.

1.1 Non-clausal Boolean Satisfiability Algorithms

The Boolean satisfiability (SAT) problem decides whether a given Boolean formula is satisfiable or unsatisfiable. The SAT problem is of central importance in various areas of computer science, including theoretical computer science, hardware and software verification, and artificial intelligence. The SAT problem is NP-complete [66] and no provably efficient algorithms are known for it. However, there have been significant (empirical) improvements [108, 117, 77] in the capacity of SAT solvers over the past decade. SAT solvers are now used routinely in many hardware verification techniques such as bounded model checking [42], k-induction [130], interpolation [112], abstraction-refinement [53, 86, 115, 103, 87]. Many software verification and static analysis tools such as CBMC [64], F-Soft [90], SATABS [62], SATURN [143], Calysto [31] rely on fast Boolean satisfiability solvers as well.

Many SAT solvers have been developed, most employing some combination of two main strategies: the Davis-Putnam-Logemann-Loveland (DPLL) search [70, 71] and heuristic local search [110]. Heuristic local search techniques are not guaranteed to be *complete*, that is, they are not guaranteed to find a satisfying assignment if one exists or prove unsatisfiability. As a result, complete SAT solvers are based almost exclusively on the DPLL search. Some well-known complete SAT solvers are GRASP [108], SATO [145], zChaff [117], BerkMin [84], Siege [18], MiniSat [77, 8], RSat [14], PicoSAT [13, 41]. From now on we will focus only on complete SAT solvers.

Most state-of-the-art SAT procedures require the input formula to be in conjunctive normal form (CNF). The design and implementation of SAT solvers becomes much easier if the input formulas are restricted to CNF. Given a truth assignment σ to a subset of variables occurring in a formula, a *Boolean constraint propagation (BCP)* algorithm determines if σ falsifies the given formula, else it provides the set of *implied assignments (unit literals)*. Modern SAT solvers spend about 80%-90% of the total time during the BCP steps. For formulas in CNF, BCP can be carried out very efficiently using the *two-watched literal* scheme [117].

While most DPLL based SAT solvers operate on CNF, there has been work on applying DPLL directly to circuit [79, 106, 137] representations. In [79] a hybrid SAT solver is described where the original formula is processed in circuit form, and learned clauses are processed separately in CNF. The circuit-based BCP is implemented by means of a lookup table. The lookup table determines the next state of a gate (or its inputs) based upon the current value of its inputs and output. The BCP on learned clauses uses the two-watched literal scheme [117]. In [137] a watched literal scheme is proposed for efficient BCP on a given circuit.

Typical formulas generated by the industrial applications are not necessarily in CNF. We refer to these formulas as *non-clausal* formulas. In order to check the satisfiability of a non-clausal formula ϕ using a CNF based SAT solver, ϕ needs to be converted to CNF. This is done by introducing new variables [138, 124]. The result is a CNF formula ϕ' which is equi-satisfiable to ϕ and is polynomial in the size of ϕ. This is the most common way of converting ϕ to a CNF formula. Conversion of a non-clausal formula to a CNF formula destroys the initial structure of the formula, which can be crucial for efficient satisfiability checking. The advantage of introducing new variables to convert ϕ to ϕ' is that it can allow for an exponentially shorter proof than is possible by completely avoiding the introduction of new variables [98]. However, the translation from ϕ to ϕ' also introduces a large number of new variables and clauses, which can potentially increase the overhead during the BCP steps and make the decision heuristics less effective. In order to reduce this overhead modern CNF SAT solvers use pre-processing techniques that try to eliminate certain variables and clauses [75]. The disadvantage with pre-processing is that it does not always lead to improvement in the SAT solver performance. It can also fail on large examples due to significant memory overhead [1].

[1] In SAT competitions the solvers disable pre-processing when the problem has more than a few million clauses to avoid running out of memory.

1.1.1 Outline of Our Results

A Boolean formula is in *negation normal form (NNF)* iff it contains only the Boolean connectives $\wedge$ (AND), $\vee$ (OR) and $\neg$ (NOT) and the scope of each occurrence of $\neg$ is a Boolean variable.

We propose a new SAT solving framework based on a representation known as *vertical-horizontal path form* (vhpform) due to Peter Andrews [29, 30]. The vhpform is a two-dimensional representation of formulas in NNF. We represent the vhpform of a given NNF formula in the form of two graphs called *vpgraph* and *hpgraph*. The vpgraph encodes the disjunctive normal form and the hpgraph encodes the conjunctive normal form of a given NNF formula. The size of these graphs is linear in the size of the given formula. We develop two non-clausal SAT algorithms that use the vpgraph and the hpgraph of a given formula.

The first algorithm is based on the idea of General Matings [29]. A path in a vpgraph starting from a root node and ending at a leaf node is called a *vertical path*. Each vertical path corresponds to a term (conjunction of literals) in a DNF representation of a given formula. At a high level our search algorithm enumerates all possible *vertical paths* in the vpgraph of a given formula until a vertical path is found that does not contain two opposite literals. If such a path is found the given formula is satisfiable. If every vertical path contains two opposite literals, then the given formula is unsatisfiable. The number of vertical paths can be exponential in the size of a given formula. Thus, the key challenge in obtaining an efficient SAT solver based on this method is to prevent the explicit enumeration of vertical paths as much as possible. We develop new techniques for preventing the explicit enumeration of vertical paths.

A path in an hpgraph starting from a root node and ending at a leaf node is called a *horizontal path*. Each horizontal path corresponds to a clause (disjunction of literals) in a CNF representation of a given formula. The hpgraph provides a compact encoding of all clauses present in a given NNF formula. The second algorithm applies the DPLL algorithm to the hpgraph representation of a given formula. The main challenge in this algorithm is to efficiently perform Boolean constraint propagation (BCP) on the hpgraph representation. We generalize the idea of the two-watched literal scheme used in CNF SAT solvers, in order to efficiently carry out BCP on hpgraphs. We evaluated the new solver on a large collection of non-clausal benchmarks drawn from bounded model checking, k-induction, equivalence checking, and software verification. The new solver is competitive with current state-of-the-art solvers in terms of run time and number of problems solved.

We refer to our algorithms as *non-clausal* SAT algorithms as they do not require the conversion of a given formula to CNF.

1.1.2 Comparison with Related Work

The key differences between existing work and our work are as follows:

1. Unlike heuristic local search based techniques, we propose complete SAT solvers.

2. Our algorithms do not operate on the circuit or CNF representation of a

given formula/circuit. In our approach a given formula/circuit is converted to an equi-satisfiable *negation normal form* (NNF) formula. The NNF formula is then represented in the form of two graphs called the vpgraph and hpgraph. These graphs are used in our SAT algorithms.

3. Our solvers handle formulas containing $\wedge, \vee, \neg$ operators and no structure sharing directly, without introduction of new variables. Observe that these formulas can easily be converted to NNF by pushing the negations to the variables using DeMorgan's laws. The existing CNF and circuit based SAT solvers require introduction of new variables for each intermediate gate or sub-formula.

4. We are also able to handle formulas with structure sharing or formulas containing other operators such as if-then-else (ITE), iff ($\Leftrightarrow$), xor ($\oplus$) operators. This is done by converting these formulas to NNF formulas as described in chapter 2. The conversion to NNF may require addition of new variables. Let V_{NNF} and V_{CNF} denote the number of new variables introduced when converting a given formula/circuit to NNF and CNF , respectively. We provide empirical justification that V_{NNF} is usually much smaller than V_{CNF}, sometimes by an order of magnitude.

5. There is also a crucial difference between the General Matings based algorithm and the DPLL algorithm. In DPLL the search space is the set of all possible assignments to the Boolean variables, whereas in General Matings the search space is the set of all possible vertical paths in the vpgraph of a

given formula. To the best of our knowledge there is no direct relationship between these search spaces.

1.2 Techniques for Word-Level Verification

Most hardware design is done at a high level of abstraction, e.g., using the *register transfer level (RT-level or RTL)*, or even at the *system level*. An RTL design describes a digital circuit in terms of data flow between registers, which store information between clock cycles in a digital circuit. The RT-level of a hardware description language such as Verilog is very similar to a software program with features for hardware design such as *bit-vectors*. Most formal verification tools used in the hardware industry convert a high level RTL design to a low level design, usually a *netlist*, for verification. A netlist is a description of a hardware design using *gates* (combinational elements) and *latches* (state-holding elements). Verification at the netlist level can be more difficult, as the high level structure of an RTL program is lost during the conversion to a netlist. For example, a multiplication operator in an RTL program gets replaced by a multiplier circuit in the netlist. This can make the verification at the netlist level less scalable.

Fig. 1.1 (a) shows the various levels of abstraction for hardware design. The ease of design increases as we move up from the netlist level to the system level. Fig. 1.1 (b) shows that hardware verification tools convert (*synthesize*) a high-level design to a netlist for verification. As argued earlier, verification at the netlist level can be more difficult, and thus, there is a need for verification techniques that

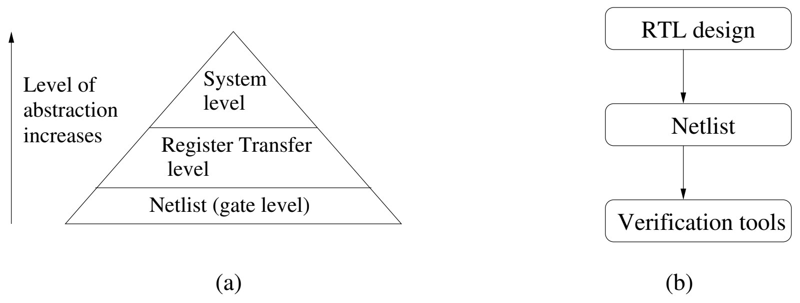


Figure 1.1: (a) Various levels of abstraction for hardware design. (b) Existing formal verification tools convert a design to a netlist.

operate directly at the RT-level or system-level. Such techniques are also referred to as *word-level* verification techniques.

1.2.1 Model Checking and Abstraction

Model checking [58, 60] is an automatic technique for the verification of finite-state concurrent systems. It has been used successfully in practice to verify complex circuit designs and communication protocols. Model checking systematically explores the state space of a given design and checks that each reachable state satisfies the property of interest. If the design fails to satisfy a desired property, the process of model checking produces a *counterexample* that demonstrates a behavior that falsifies the property. By making use of *symbolic* algorithms [52, 111, 42, 26] based on Binary Decision Diagrams (BDDs) [51] or fast satisfiability solvers (SAT solvers) [108, 117, 8], current model checkers can scale to systems with a large number of states.

In industrial hardware designs the number of states is extremely large. This results in a state explosion problem during model checking even when symbolic model checking algorithms are used. One principal method in state space reduction is *abstraction*. Abstraction techniques reduce the state space by mapping the set of states of the actual, concrete system to an abstract, and smaller, set of states in a way that preserves the relevant behaviors of the system.

Many formal verification tools use abstraction techniques that produce a *conservative* over-approximation of the concrete system. This implies that if the abstraction satisfies a given property, the property also holds on the original concrete system. The drawback of the conservative abstraction is that when model checking of the abstraction fails, it may produce an *abstract counterexample* that does not correspond to any concrete counterexample. This is usually called a *spurious counterexample* [55].

In order to check if an abstract counterexample is spurious, the abstract counterexample is simulated on the concrete program. This is called the *simulation* step. As in *bounded model checking* (BMC) [42], the concrete transition relation for the design and the given property are jointly unwound to obtain a Boolean formula. The number of unwinding steps is given by the length of the abstract counterexample. The Boolean formula is then checked for satisfiability using a SAT procedure. If the instance is satisfiable, the counterexample is real and the procedure terminates. If the instance is unsatisfiable, the abstract counterexample is spurious, and *abstraction refinement* has to be performed.

The basic idea of abstraction refinement techniques [102, 55, 61, 33] is to create a new abstract model that contains more detail in order to prevent the spu-

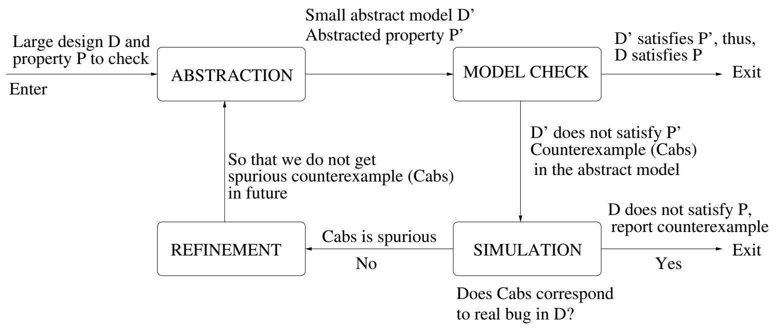


Figure 1.2: Counterexample Guided Abstraction and Refinement (CEGAR) Loop.

rious counterexample. This process is iterated until the property is either proved or disproved. It is known as the *Counterexample Guided Abstraction Refinement* framework, or CEGAR for short [55]. The CEGAR loop is shown in Fig. 1.2.

1.2.2 Abstraction Techniques for Circuits

Most model checkers used in hardware verification operate on a low level design, usually a netlist. At the netlist level, a commonly used abstraction technique is *localization reduction* [102, 142, 87]. The abstract model is created from the given circuit by removing a large number of latches together with the logic required to compute their next state. The latches that are removed are called the *invisible latches*. The latches remaining in the abstract model are called *visible latches*. For example, the initial abstract model can be created by making the latches present in the property visible, and the rest invisible. The refinement is done by moving more latches from the set of invisible latches to the set of visible latches. The refinement step is usually based on the analysis of a spurious counterexample.

A *proof-based* approach is followed in [115, 86] where a proof of unsatisfiability produced by a SAT solver is used to refine the abstraction. The advantage of the proof-based approaches is that all counterexamples upto a given depth are eliminated from the abstract model at each refinement step.

McMillan [112] describes a SAT-based method for finite-state model checking based on the use of *interpolants*. In [112] the idea of interpolation is combined with bounded model checking to obtain an over-approximate image operator. This allows obtaining over-approximations of the reachable set of states without using the costly image computation (existential quantification) operations. The use of interpolation helps the verification procedure focus only on the parts of design that are relevant to proving the property.

1.2.3 Outline of Our Results

As described above there are numerous abstraction techniques available for verification at the netlist level. However, very few abstraction techniques are known for verification at the word-level. Since word-level hardware designs are similar to software, we propose the use of abstraction algorithms that have been devised for software verification.

In the software domain, one successful abstraction technique for large systems is *predicate abstraction* [85]. It abstracts data by only keeping track of certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. Predicate abstraction of ANSI-C programs in combination with CEGAR loop was introduced by Ball and Rajamani [33] and promoted by the success of the SLAM project. The goal of the SLAM project is to verify that Windows device drivers obey API conventions.

We apply predicate abstraction in combination with CEGAR loop for verifying RTL Verilog programs. There are two challenges when applying predicate abstraction to circuits: 1) The computation of the abstract model in presence of a large number of predicates, and 2) the discovery of suitable word-level predicates for abstraction refinement. We address these problems as part of this thesis.

1.2.4 Comparison with Related Work

While localization reduction is a special case of predicate abstraction, predicate abstraction can result in a much smaller abstract model. As an example, assume a circuit contains two registers, each encoding a number. Predicate abstraction can keep track of a numerical relation between the two numbers using a single predicate, and thus, using a single state bit in the abstract model. In contrast, localization reduction typically turns all bits of the two registers into visible latches, and thus, the abstraction is identical to the original model.

Clarke et al. [63] introduce a SAT-based technique for predicate abstraction of netlist level circuits. The use of a SAT solver like zChaff [117] in order to perform the abstraction allows precise modeling of bit-vector semantics. However, their approach suffers from two drawbacks. 1) Each transition in the abstract model is computed by a separate run of the SAT solver. Thus, the learning done by a SAT solver in the form of conflict clauses is lost when computing other transitions in the abstract model. 2) If refinement becomes necessary, only bit-level predicates are introduced. This method of refinement closely resembles refinement techniques for localization reduction.

Andraus et al. [28] present a scheme for automatic abstraction of behavioral RTL Verilog to the CLU language [50]. The CLU language allows modeling using terms, uninterpreted functions, equality, lambda expressions, and counters. In order to remove spurious behaviors from the abstract model a refinement procedure is described in [27]. The techniques in [28, 27] were shown to be useful in context of microprocessor correspondence checking. The techniques we propose are different from those in [28, 27] and are geared towards property (assertion) checking of hardware designs.

A *Pre-image computation* generates a set of states from which it is possible to reach a given set of states with one transition. It is a basic operation in model checking [58] and target enlargement approaches [39]. The idea of computing a pre-image is the same as computing the weakest precondition of a given set of states, although the latter term is more commonly used in software verification. Most existing hardware model checkers compute the pre-image at the netlist level and represent it symbolically using BDDs. As in software verification our use of weakest preconditions or pre-images is at the word (expression) level.

We use weakest pre-conditions for discovering new predicates for refinement. This technique can lead to too many refinement iterations or may not even terminate in some cases, for example, when we need to track the value of an n bit counter c in an abstract model precisely. In this case refinement using weakest preconditions can lead to 2^n iterations, where each iteration discovers a predicate of the form $c = v$, $0 \leq v \leq 2^n - 1$. In localization reduction the value c can be tracked precisely by making each bit in c a visible latch. It is possible to get the benefits of localization reduction in our technique as well by adding $c[0], \ldots, c[n-1]$ as predicates. The combination of predicate abstraction and localization reduction is studied in detail by Wang et al. [140].

Lahiri and Bryant [104] propose an extension to predicate abstraction that uses predicates with free (index) variables. This allows verification of safety properties of unbounded systems. In our context, indexed predicates can be useful when dealing with memories or input variables. The predicate discovery heuristics described in [104] can be used in our context.

An alternative technique for discovering new predicates is based on Craig interpolation [113, 89]. This technique is used in a state-of-the-art software model checker BLAST [2]. In order to apply this idea to circuits, an interpolating theorem prover for bit-vector logic [49, 38, 32, 107, 82, 48] is required. At present, it is not known how to build a practical interpolating theorem prover for bit-vector logic. We have developed an efficient interpolation algorithm for conjunctions of *linear modular equations (linear congruences).* Our algorithm handles the interpolation problem for a subset of bit-vector logic.

1.3 Craig Interpolation for Subsets of Integer Linear Arithmetic

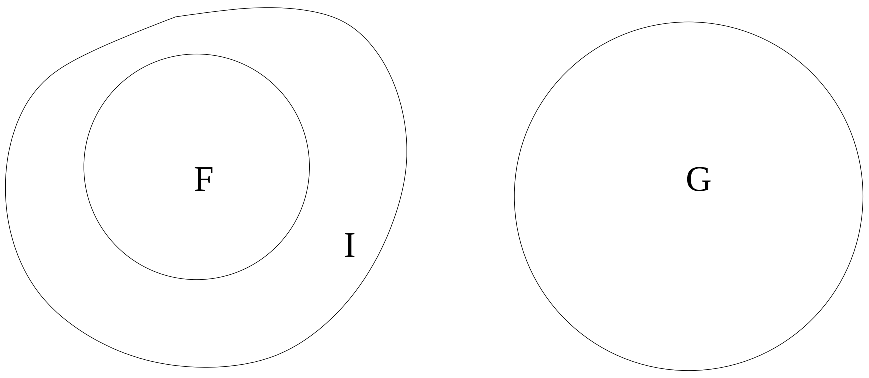


Figure 1.3: Formulas F, G, I represented as sets. $F \wedge G$ is unsatisfiable and I represents an interpolant for (F, G).

The use of Craig interpolation [67] has led to powerful hardware [112] and software [89, 114] model checking techniques. Given two formulas F, G such that $F \wedge G$ is unsatisfiable, a Craig interpolant for the pair (F, G) is a formula I with the following properties: 1) $F \Rightarrow I$, 2) $I \wedge G$ is unsatisfiable, and 3) I refers only to the common variables of F and G. One can view a formula as the set of states that make the formula true. Figure 1.3 shows that the sets representing formulas F, G are disjoint. The set representing the interpolant I for (F, G) is an over-approximation (superset) of F and is disjoint from G.

In [112] the idea of interpolation is used for obtaining over-approximations of the reachable set of states without using the costly image computation (existential quantification) operations. In [89, 99] interpolants are used for finding the

```
void main()
{
  int x=1,y=2;
  while(1)
  {
    x = x +3*nondet();
    y = y +6*nondet();
    if (x+y==2)
      ERROR: ;
  }
}
```

Figure 1.4: A C program with an unreachable ERROR label.

right set of predicates in order to rule out *spurious counterexamples* in a CEGAR framework. An interpolating theorem prover performs the task of finding the interpolants. Such provers are available for various theories such as propositional logic, rational and real linear arithmetic and equality with uninterpreted functions [113, 144, 100, 99, 128, 101, 54].

1.3.1 Motivating Example

Consider the C code in Fig.1.4. The function call `nondet()` returns a random integer. We are interested in checking the reachability of the `ERROR` label. Intuitively, the `ERROR` label is unreachable because $x+y$ is a multiple of 3 when the condition of the `if` statement is checked. Existing weakest precondition based or interpolation based model checkers are not able to find the right predicates in order to show that the ERROR label is unreachable. In this example the right predicate is $x+y \equiv 0\ (mod\ 3)$, that is, $x+y$ is a multiple of 3. We have developed new interpolation algorithms that are effective at discovering *modular/divisibility predicates*, such as $x + y \equiv 0\ (mod\ 3)$, from spurious counterexamples.

1.3.2 Outline of Our Results

Efficient algorithms are known for computing interpolants in rational and real linear arithmetic [113, 128, 54]. Linear arithmetic formulas where all variables are constrained to be integers are said to be formulas in *(pure) integer linear arithmetic* or $LA(\mathbb{Z})$, where $\mathbb{Z}$ is the set of integers. There are no known efficient algorithms for computing interpolants for formulas in $LA(\mathbb{Z})$. This is expected because checking the satisfiability of conjunctions of atomic formulas in $LA(\mathbb{Z})$ is itself NP-hard. We show that for various *subsets* of $LA(\mathbb{Z})$ one can compute proofs of unsatisfiability and interpolants in polynomial time. We demonstrate the utility of the proposed interpolation algorithms for discovering *modular/divisibility* predicates in a counterexample guided abstraction refinement (CEGAR) framework. This has enabled verification of simple programs that cannot be checked using existing CEGAR based model checkers.

1.4 Thesis Outline

- In chapter 2 we discuss the conversion of Boolean circuits to NNF formulas and the vertical-horizontal path form (vhpform) representation of NNF formulas. We describe how to represent vhpform in form of two graphs called vpgraph and hpgraph. This chapter forms the basis of our non-clausal SAT algorithms described in chapters 3 and 4.

- We describe our General Matings based SAT algorithm [91] in chapter 3. The techniques for search space pruning, learning, and non-chronological backtracking are presented. Experimental evaluation of the solver is presented.

- In chapter 4 we describe our DPLL based SAT algorithm. We present an algorithm for carrying out Boolean constraint propagation on hpgraphs by using a generalization of the two-watched literal scheme and the vpgraph. We present an experimental evaluation of the solver.

- We present techniques for verifying register transfer level (RTL) Verilog using predicate abstraction and counterexample guided abstraction refinement (CEGAR) loop [95, 96, 97, 23] in chapter 5.

- In chapter 6 we present our results on Craig interpolation for subsets of integer linear arithmetic [92].

- We conclude in chapter 7 with directions for future research.

Chapter 2

Graph Based Representations for Non-Clausal SAT Solving

The problem of Boolean (propositional) satisfiability (SAT) is of central importance in various areas of computer science, including theoretical computer science, artificial intelligence, and hardware/software design and verification. Most state-of-the-art SAT procedures are variations of the Davis-Putnam-Logemann-Loveland (DPLL) [70, 71] algorithm and require the input formula to be in conjunctive normal form (CNF). Typical formulas arising in practice are *non-clausal*, that is, not in CNF. Converting a non-clausal formula to CNF introduces overhead in form of new variables and may destroy the initial structure of the formula, which can be crucial in efficient satisfiability checking.

We propose a new Boolean SAT solving framework based on a representation known as *vertical-horizontal path form* (vhpform) due to Peter Andrews [29, 30].

We develop two non-clausal SAT algorithms that use the vhpform of a given formula. We describe the vhpform representation in this chapter. In chapters 3, 4 we describe our SAT algorithms that use the vhpform of a given formula.

A Boolean formula is in *negation normal form (NNF)* iff it contains only the Boolean connectives $\wedge$ (and), $\vee$ (or) and $\neg$ (not), the scope of each occurrence of $\neg$ is a Boolean variable. We also require that there is no *structure sharing* in a NNF formula, that is, output from a gate acts as input to atmost one gate. That is, a NNF formula is tree-like as opposed to a circuit which can be DAG-like.

The vhpform is defined for formulas in NNF form. Most Boolean circuits obtained in practice are not in NNF form. The conversion of Boolean circuits to NNF formulas is described in the next section.

2.1 Conversion of Boolean Formulas/Circuits to Negation Normal Form Formulas

In our work Boolean circuits are converted to NNF formulas in two stages. The first stage re-writes other operators (xor, iff, implies, if-then-else) in terms of $\wedge$, $\vee$, $\neg$ operators. For example, $\phi_1 \Leftrightarrow \phi_2$ (ϕ_1 iff ϕ_2) is written as $(\neg\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \neg\phi_2)$. In order to avoid a blowup in the size of the resulting formula we allow sharing of sub-formulas. Thus, the first stage produces a formula containing $\wedge, \vee, \neg$ gates, possibly with structure sharing. The second stage gets rid of the structure sharing in order to obtain a NNF formula. This is done by introduction of new variables. We introduce a new variable for each gate that is different from a $\neg$ (not) gate and

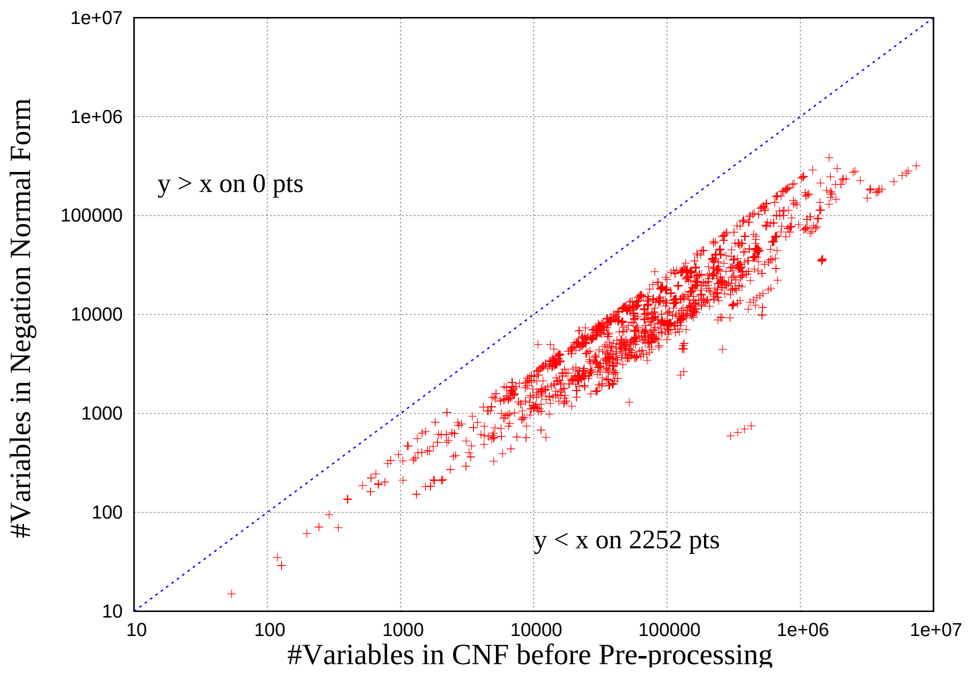


Figure 2.1: Comparing number of variables in NNF formulas on y-axis and number of variables in CNF formulas on x-axis.

has a fanout greater than one. Observe that the conversion of a Boolean circuit to a NNF formula can be done in linear time in the size of the Boolean circuit.

In Figure 2.1 we compare the number of variables in the NNF and CNF representations of a collection of 2541 industrial non-clausal benchmarks (Boolean circuits). The CNF form was obtained by means of the standard Tseitin translation [138, 124]. We can see that the NNF forms have $5-10$ times fewer variables. Modern CNF SAT solvers use pre-processing techniques in order to eliminate certain variables and clauses from the input CNF formula [75]. We compare the variables in the pre-processed CNF formulas and the corresponding NNF formulas in

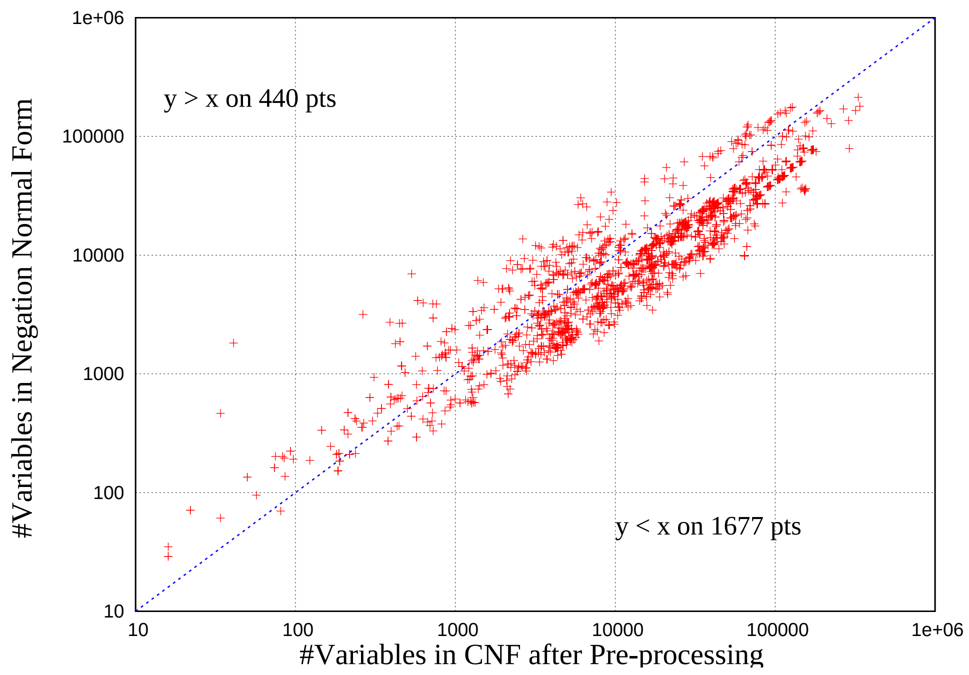


Figure 2.2: Comparing number of variables in NNF formulas on y-axis and number of variables in pre-processed CNF formulas on x-axis.

Figure 2.2. The CNF formulas were pre-processed using SatELite [75]. Observe that pre-processing is able to reduce the number of variables in the CNF formulas significantly. However, on majority ($> 70\%$) of benchmarks the CNF form still has more variables than NNF form. The fewer variables in the NNF form (without any pre-processing) motivates the need for exploring SAT solving techniques that operate on NNF directly.

In the subsequent sections and the next two chapters we assume that the input Boolean formula/circuit has been converted to a NNF formula. Given an NNF formula ϕ our SAT algorithms check the satisfiability of ϕ without introducing

$$\left[\begin{array}{c} \left[\begin{array}{c} p \vee q \\ \\ -r \\ \\ -q \end{array}\right] \vee \left[\begin{array}{c} -p \\ \\ r \vee -s \\ \\ q \end{array}\right] \end{array}\right]$$

Figure 2.3: The vhpform for the formula $(((p \vee q) \wedge \neg r \wedge \neg q) \vee (\neg p \wedge (r \vee \neg s) \wedge q))$. We show the negation of a variable by a $-$ sign.

any more new variables.

2.2 Vertical-Horizontal Path Form

The internal representation in our satisfiability solver is NNF. More specifically, we use a two-dimensional representation of a NNF formula, called *vertical-horizontal path form (vhpform)* as described in [30][1]. In this form disjunctions are written horizontally and conjunctions are written vertically. For example Fig. 2.3 shows the formula $\phi = (((p \vee q) \wedge \neg r \wedge \neg q) \vee (\neg p \wedge (r \vee \neg s) \wedge q))$ in vhpform. We define two types of paths in the vhpform of a given formula.

Vertical path: A vertical path through a vhpform is a sequence of literals in the vhpform that results by choosing either the left or the right scope for each occurrence of $\vee$. For the vhpform in Fig. 2.3 the set of vertical paths is $\{\langle p, \neg r, \neg q \rangle$,

[1]In [30] the term *vertical path form (vpform)* is used in place of vertical-horizontal path form (vhpform). We use vertical-horizontal path form (vhpform) to emphasize the use of both vertical and horizontal paths.

$\langle q, \neg r, \neg q\rangle, \langle \neg p, r, q\rangle, \langle \neg p, \neg s, q\rangle\}$.

Horizontal path: A horizontal path through a vhpform is a sequence of literals in the vhpform that results by choosing either the upper or the lower scope for each occurrence of $\wedge$. For the vhpform in Fig. 2.3 the set of horizontal paths is $\{\langle p, q, \neg p\rangle, \langle p, q, r, \neg s\rangle, \langle p, q, q\rangle, \langle \neg r, \neg p\rangle, \langle \neg r, r, \neg s\rangle, \langle \neg r, q\rangle, \langle \neg q, \neg p\rangle, \langle \neg q, r, \neg s\rangle,$ $\langle \neg q, q\rangle\}$.

Two important results regarding satisfiability of negation normal formulas from [30] are given below. Let F be a formula in negation normal form and let σ be an assignment (σ can be a partial assignment).

Theorem 1 *σ satisfies F iff there exists a vertical path P in the vhpform of F such that σ satisfies every literal in P.*

Theorem 2 *σ falsifies F iff there exists a horizontal path P in the vhpform of F such that σ falsifies every literal in P.*

Example 1 The vhpform in Fig. 2.3 has a vertical path $\langle p, \neg r, \neg q\rangle$ whose every literal can be satisfied by an assignment σ that sets p to true and r, q to false. It follows from Theorem 1 that σ satisfies ϕ. Thus, ϕ is satisfiable. All literals in the vertical path $\langle q, \neg r, \neg q\rangle$ cannot be satisfied simultaneously by any assignment (due to opposite literals q and $\neg q$).

An assignment σ' that sets p, r to true, falsifies every literal in the horizontal path $\langle \neg r, \neg p\rangle$ in the vhpform of ϕ. Thus, from Theorem 2 it follows that σ' falsifies ϕ.

Let $\mathcal{VP}(\phi)$ and $\mathcal{HP}(\phi)$ denote the set of vertical paths and the set of horizontal paths in the vhpform of a given formula ϕ, respectively. We use $l \in \pi$ to denote the occurrence of a literal l in a vertical/horizontal path π. The following result from [30] states that the set of vertical paths encodes the DNF and the set of horizontal paths encodes the CNF of a given formula.

Theorem 3 *Let ϕ be a NNF formula.*
(a) ϕ is equivalent to the DNF formula $\bigvee_{\pi \in \mathcal{VP}(\phi)} \bigwedge_{l \in \pi} l$.
(b) ϕ is equivalent to the CNF formula $\bigwedge_{\pi \in \mathcal{HP}(\phi)} \bigvee_{l \in \pi} l$.

Our SAT algorithms operate on graph based representations of the vhpform of a given formula. We describe these graph based representations below.

2.3 Graph Based Representations

2.3.1 Graphical Encoding of Vertical Paths (Vpgraph)

A graph containing all vertical paths present in the vhpform of a NNF formula is called a *vpgraph*. Given a NNF formula ϕ, we define the vpgraph $G_v(\phi)$ as a tuple (V, R, L, E, Lit), where V is the set of nodes corresponding to all occurrences of literals in ϕ, $R \subseteq V$ is a set of root nodes, $L \subseteq V$ is a set of leaf nodes, $E \subseteq V \times V$ is the set of edges, and $Lit(n)$ denotes the literal associated with node $n \in V$. A node $n \in R$ has no incoming edges and a node $n \in L$ has no outgoing edges.

The vpgraph containing all vertical paths in the vhpform of Fig. 2.4(a) is shown in Fig. 2.4(b). For the vpgraph in Fig. 2.4(b), we have $V = \{1,2,3,4,5,6,7,8\}$,

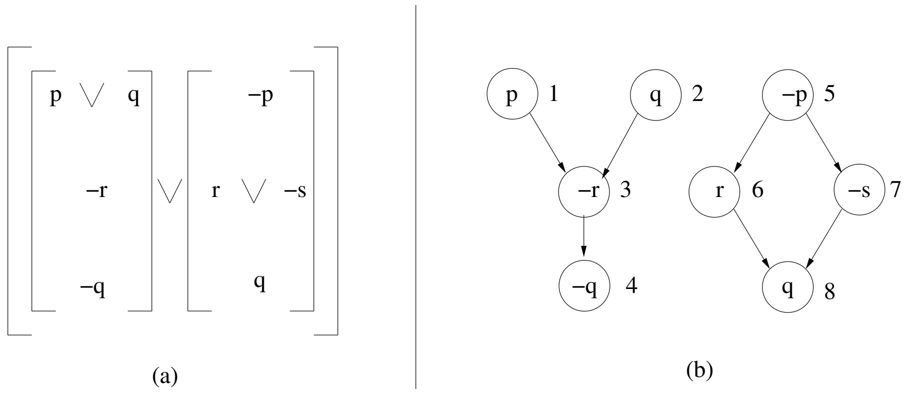


Figure 2.4: (a) The vhpform for the formula $(((p \vee q) \wedge \neg r \wedge \neg q) \vee (\neg p \wedge (r \vee \neg s) \wedge q))$ (b) the corresponding vpgraph.

$R = \{1,2,5\}$, $L = \{4,8\}$, $E = \{(1,3),(2,3),(3,4),(5,6),(5,7),(6,8),(7,8)\}$ and for each $n \in V$, $Lit(n)$ is shown inside the node labeled n in Fig. 2.4(b). Each path in the vpgraph $G_v(\phi)$, starting from a root node and ending at a leaf node, corresponds to a vertical path in the vhpform of ϕ. For example, path $\langle 1,3,4 \rangle$ in Fig. 2.4(b) corresponds to the vertical path $\langle p, \neg r, \neg q \rangle$ in Fig. 2.4(a) (obtained by replacing node n on path by $Lit(n)$). Using this correspondence one can see that the vpgraph contains all vertical paths present in the vhpform shown in Fig. 2.4(a). Given ϕ, we can construct the vpgraph $G_v(\phi) = (V,R,L,E,Lit)$ directly without constructing the vhpform of ϕ. This is done inductively as follows:

- If ϕ is a literal l, then we create a graph containing just one node fv, where fv is a fresh identifier (node number). The literal stored inside fv is set to l.

  $G_v(\phi) = (\{fv\},\{fv\},\{fv\},\emptyset,Lit)$ and $Lit(fv) = l$, fv is a fresh identifier.

- If $\phi = \phi_1 \vee \phi_2$, then the vpgraph for ϕ is obtained by taking the union of the vp-graphs of ϕ_1 and ϕ_2. Let $G_v(\phi_1) = (V_1, R_1, L_1, E_1, Lit_1)$ and $G_v(\phi_2) = (V_2, R_2, L_2, E_2, Lit_2)$. Then $G_v(\phi)$ is the union of $G_v(\phi_1)$ and $G_v(\phi_2)$.

$G_v(\phi) = (V_1 \cup V_2, R_1 \cup R_2, L_1 \cup L_2, E_1 \cup E_2, Lit_1 \cup Lit_2)$

- If $\phi = \phi_1 \wedge \phi_2$, then the vpgraph for ϕ is obtained by concatenating the vpgraph of ϕ_1 with the vpgraph of ϕ_2. Let $G_v(\phi_1) = (V_1, R_1, L_1, E_1, Lit_1)$ and $G_v(\phi_2) = (V_2, R_2, L_2, E_2, Lit_2)$. Then $G_v(\phi)$ contains all the nodes and edges in $G_v(\phi_1)$ and $G_v(\phi_2)$. But $G_v(\phi)$ has additional edges connecting leaves of $G_v(\phi_1)$ with the roots of $G_v(\phi_2)$. The set of additional edges is denoted as $L_1 \times R_2$ below. The set of roots of $G_v(\phi)$ is R_1, while the set of leaves is L_2.

  $G_v(\phi) = (V_1 \cup V_2, R_1, L_2, E_1 \cup E_2 \cup (L_1 \times R_2), Lit_1 \cup Lit_2)$

2.3.2 Graphical Encoding of Horizontal Paths (Hpgraph)

A graph containing all horizontal paths present in the vhpform of a NNF formula is called a *hpgraph*. We use $G_h(\phi)$ to denote the hpgraph of a formula ϕ. The procedure for constructing a hpgraph is similar to the above procedure for constructing the vpgraph. The difference is that the hpgraph for $\phi = \phi_1 \wedge \phi_2$ is obtained by taking the union of the hpgraphs for ϕ_1 and ϕ_2 and the hpgraph for $\phi = \phi_1 \vee \phi_2$ is obtained by concatenating the hpgraphs of ϕ_1 and ϕ_2.

The hpgraph containing all horizontal paths in the vhpform in Fig. 2.5(a) is shown in Fig. 2.5(b). For the hpgraph in Fig. 2.5(b), we have $V = \{1,2,3,4,5,6,7,8\}$, $R = \{1,3,4\}$, $L = \{5,7,8\}$, $E = \{(1,2),(2,5),(2,6),(2,8),(3,5),(3,6),(3,8),(4,5),(4,6),(4,8),(6,7)\}$ and for each $n \in V$, $Lit(n)$ is shown inside the node labeled n.

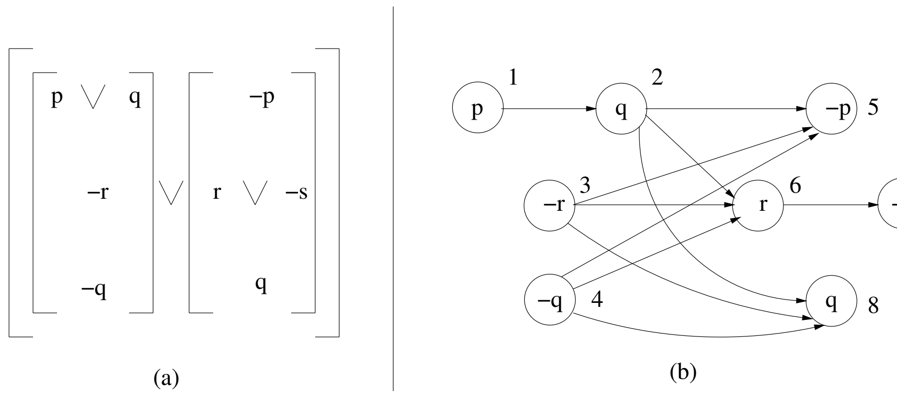


Figure 2.5: (a) The vhpform for the formula $(((p \vee q) \wedge \neg r \wedge \neg q) \vee (\neg p \wedge (r \vee \neg s) \wedge q))$ (b) the corresponding hpgraph.

It can be shown by induction that the vpgraph and hpgraph of a NNF formula are directed acyclic graphs (DAGs). One can also represent vpgraph and hpgraph as *directed series-parallel* graphs. Series-parallel graphs have been widely studied and many problems that are NP-complete for general graphs can be solved in linear time for series-parallel graphs [135].

The construction of vpgraph/hpgraph can be done in $O(k)$ time/space where k is the size of the given NNF formula. We refer the reader to appendix A for more details.

When constructing an hpgraph/vpgraph from a NNF formula ϕ each literal in ϕ gets represented as a new node in the hpgraph and vpgraph of ϕ. We assume that the node number corresponding to each literal l in ϕ is the same in the hpgraph and the vpgraph of ϕ. Thus, the set of nodes in the hpgraph and vpgraph are identical.

In the following we define some terms that will be used in the next two chapters. When the distinction between $G_h(\phi)$ and $G_v(\phi)$ is not required we drop the subscripts and use $G(\phi)$.

2.3.3 Terminology

Let $G(\phi) = (V,R,L,E,Lit)$ denote a vpgraph or a hpgraph.

Definition 1 *A path $\pi = \langle n_0,\ldots,n_k\rangle$ in $G(\phi)$ is said to be a* **r-path (rooted path)** *iff it starts with a root node ($n_0 \in R$). Formally, $\pi = \langle n_0,\ldots,n_k\rangle$ is a r-path iff $n_0 \in R$ and $(n_i,n_{i+1}) \in E$ for all $0 \leq i < k$.*

In Fig. 2.4(b), $\langle 2,3\rangle$ is a r-path while $\langle 3,4\rangle$ is not a r-path.

Definition 2 *A path $\pi = \langle n_0,\ldots,n_k\rangle$ in $G(\phi)$ is said to be a* **rl-path** *iff it starts at a root node and ends at a leaf node. Formally, $\pi = \langle n_0,\ldots,n_k\rangle$ is a rl-path iff $n_0 \in R$, $n_k \in L$ and $(n_i,n_{i+1}) \in E$ for all $0 \leq i < k$.*

In Fig. 2.4(b), both $\langle 2,3,4\rangle$, $\langle 5,6,8\rangle$ are rl-paths, but $\langle 3,4\rangle$ is not a rl-path.

There is a one-to-one correspondence between the rl-paths in $G_v(\phi)$ and the vertical paths in the vhpform of ϕ. There is a similar one-to-one correspondence between the rl-paths in $G_h(\phi)$ and the horizontal paths in the vhpform of ϕ. For example, path $\langle 1,2,6,7\rangle$ in Fig. 2.5(b) corresponds to the horizontal path $\langle p,q,r,\neg s\rangle$ in Fig. 2.5(a). The following corollary adapts Theorem 3 to the graphical representations.

Corollary 1 *Let π denote an rl-path and n denote a node on π.*
(a) ϕ is equivalent to the DNF formula $\bigvee_{\pi\in G_v(\phi)} \bigwedge_{n\in\pi} Lit(n)$.
(b) ϕ is equivalent to the CNF formula $\bigwedge_{\pi\in G_h(\phi)} \bigvee_{n\in\pi} Lit(n)$.

We will find it convenient to think of an rl-path π in $G_h(\phi)$ as a clause $\bigvee_{n\in\pi} Lit(n)$ in the CNF representation of ϕ. This is justified by the above corollary. Similarly, one can think of an rl-path π in $G_v(\phi)$ as a term (cube) $\bigwedge_{n\in\pi} Lit(n)$ in the DNF representation of ϕ.

2.4 Chapter Summary

We presented a two-dimensional representation of NNF formulas called vertical-horizontal path form (vhpform). The vhpform of an NNF formula contains vertical and horizontal paths. A vertical path is like a cube (term) in the DNF representation of a given formula, while a horizontal path is like a clause in the CNF representation of a given formula. The vpgraph encodes all vertical paths and the hpgraph encodes all horizontal paths. Both vpgraph and hpgraph can be obtained in linear time in the size of the given NNF formula.

Typical Boolean circuits arising in practice are not in NNF form. Such circuits can be converted to NNF form efficiently by introducing new variables. The NNF of a circuit is usually more succinct than the (pre-processed) CNF of the circuit in terms of number of variables.

Chapter 3

General Matings based SAT Solver

General Matings is a theorem proving technique due to Andrews [29]. It is closely related to the *Connection method* discovered independently by Bibel [40]. Theorem provers based on these techniques have been used successfully in higher order theorem proving [21]. We use the General Matings idea to build a SAT solver for satisfiability problems arising in practice.

Theorem 1 (Section 2.2) forms the basis of our General Matings based SAT solver called `SatMate`. The idea is to check the satisfiability of a given NNF formula by examining the vertical paths in its vpgraph. At a high level our search algorithm enumerates all possible vertical paths in the vpgraph of a given formula until a vertical path is found that does not contain two opposite literals. If such a path is found the given formula is satisfiable. If every vertical path contains two opposite literals, then the given formula is unsatisfiable. The number of vertical paths can be exponential in the size of a given formula. Thus, the key challenge in obtaining an efficient SAT solver based on this method is to prevent the enumeration of vertical paths as much as possible. We develop several new techniques for preventing the enumeration of vertical paths. Our contributions can be summarized as follows.

3.1 Contributions

- Our solver employs a combination of both vertical and horizontal path exploration for efficient SAT solving. The choice of which variable to assign next (*decision making*) is made using the vertical paths, which are similar to the terms (conjunction of literals) in the DNF of a given formula. Conflict detection is aided by the use of horizontal paths, which are similar to the clauses (disjunction of literals) in the CNF of a given formula.
- We show how to adapt the techniques found in the current state-of-the-art SAT solvers to our algorithm. We describe how to perform *search space pruning*, *conflict driven learning*, and *non-chronological backtracking* by using the vertical paths and horizontal paths in the vhpform of a given formula.

3.2 Preliminaries

Let $G(\phi) = (V,R,L,E,Lit)$ denote a vpgraph or hpgraph.

Definition 3 *Two nodes $n_1, n_2 \in V$ are said to be* **conflicting** *iff $Lit(n_1) = \neg Lit(n_2)$.*

In the vpgraph shown in Fig. 3.1(a), nodes 2,4 are conflicting.

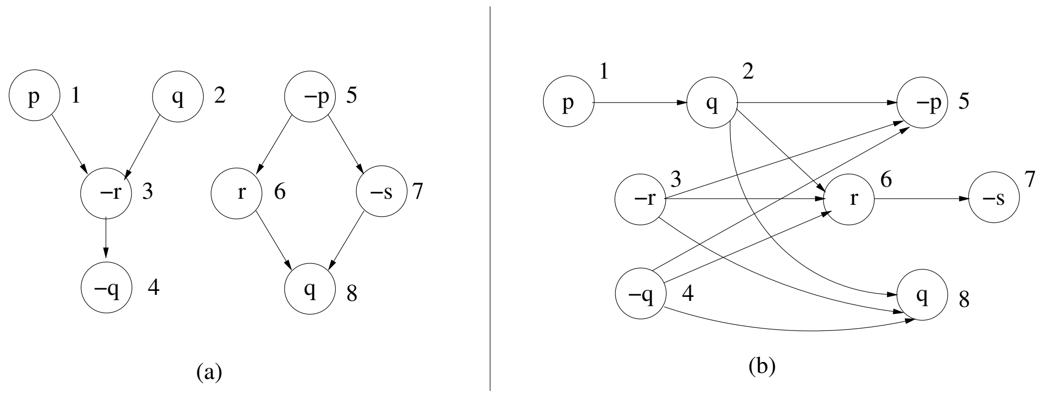


Figure 3.1: (a) The vpgraph for $(((p \vee q) \wedge \neg r \wedge \neg q) \vee (\neg p \wedge (r \vee \neg s) \wedge q))$ (b) the corresponding hpgraph.

Definition 4 *We say an assignment* σ **satisfies (falsifies) a node** $n \in V$ *iff* σ *satisfies (falsifies)* $Lit(n)$.

An assignment that sets q to true satisfies nodes 2, 8 and falsifies node 4 in Fig. 3.1(a).

Definition 5 *We say an assignment* σ satisfies (falsifies) a path $\pi \in G(\phi)$ *iff* σ *satisfies (falsifies) every node on* π.

For example, in Fig. 3.1(a) path $\langle 5, 6, 8 \rangle$ is satisfied by an assignment which sets p to false and r, q to true. The same path is falsified by an assignment which sets p to true and r, q to false.

Definition 6 *We say that a path* $\pi \in G$ *is* satisfiable *iff there exists an assignment which satisfies* π.

In Fig. 3.1(a), path $\langle 5,6,8 \rangle$ is satisfiable, while the path $\langle 2,3,4 \rangle$ is not satisfiable due to conflicting nodes 2,4.

Recall, that an rl-path in a vpgraph $G_v(\phi)$ corresponds to a vertical path in the vhpform of ϕ. Similarly, an rl-path in a hpgraph $G_h(\phi)$ corresponds to a horizontal path in the vhpform of ϕ. The following corollaries adapt Theorem 1 and Theorem 2 (Section 2.2) to the graph representations of the vhpform of a given formula ϕ.

Corollary 2 *An assignment σ satisfies ϕ iff there exists a rl-path π in $G_v(\phi)$ such that σ satisfies π.*

Corollary 3 *An assignment σ falsifies ϕ iff there exists a rl-path π in $G_h(\phi)$ such that σ falsifies π.*

The following corollary is a re-statement of corollary 2.

Corollary 4 *ϕ is satisfiable iff there exists a rl-path π in $G_v(\phi)$ which is satisfiable.*

The following corollary connects the notion of conflicting nodes with the satisfiability of a path.

Corollary 5 *A path π in $G(\phi)$ is satisfiable iff no two nodes on π are conflicting.*

Discovery of unit literals from hpgraph: Modern SAT solvers operating on a CNF representation employ a *unit literal rule* for efficient Boolean constraint propagation. The unit literal rule states that if all but one literal of a clause are set to false, then the un-assigned literal in the clause must be set to true under the current assignment. In our context the input formula is not necessarily represented in CNF, however, it is still possible to obtain the unit literal rule via the use of the hpgraph of a given formula. The following claim states the unit literal rule for the non-clausal formulas.

Corollary 6 *If an assignment σ falsifies all but one node (say n) on an rl-path π in $G_h(\phi)$ and $Lit(n)$ is not already assigned by σ, then $Lit(n)$ must be set to true under the current assignment σ in order to obtain a satisfying assignment.*

Intuitively, each rl-path in the hpgraph corresponds to a clause in the CNF of a given formula (Corollary 1 (b)). Thus, at least one literal from each rl-path in $G_h(\phi)$ must be satisfied in order to obtain a satisfying assignment.

Example 2 Consider the hpgraph shown in Fig. 3.1(b) and an assignment σ which sets p, q to false and s to true. σ falsifies all but node 6 on the rl-path $\langle 1,2,6,7 \rangle$ in the hpgraph. It follows from Corollary 6 that $Lit(6)$ which is r must be set to true under σ.

We give a high level description of our General Matings based solver called `SatMate` below.

3.3 Top Level Algorithm Used in SatMate

In order to check the satisfiability of a NNF formula ϕ, we obtain a vpgraph $G_v(\phi)$. From Corollary 4 it follows that ϕ is satisfiable iff $G_v(\phi)$ has a satisfiable rl-path. At a high level our search algorithm enumerates all possible rl-paths until a satisfiable rl-path is found. If no satisfiable rl-path is found, then ϕ is unsatisfiable. For

DNF (or DNF-like) formulas the number of rl-paths in vpgraph is small, linear in the size of the formula, and therefore the basic search algorithm is efficient. However, for formulas that are not in DNF form, the algorithm of just enumerating all rl-paths in $G_v(\phi)$ does not scale. We have adapted several techniques found in modern SAT solvers such as *search space pruning*, *conflict driven learning*, *non-chronological backtracking* to make the search efficient.

Algorithm 3.1 Searching a vpgraph for a satisfiable rl-path.

```
Input: vpgraph $G_v(\phi) = (V, R, L, E, Lit)$ and hpgraph $G_h(\phi) = (V', R', L', E', Lit')$
Output: If $G_v(\phi)$ has a satisfiable rl-path return SAT, else return UNSAT
 1: $st \leftarrow R$ {push all roots in $G_v(\phi)$ on stack $st$}
 2: $\sigma \leftarrow \emptyset$ {initial truth assignment is empty}
 3: $\forall n \in V : mrk(n) \leftarrow false$ {all nodes are un-marked to start with}
 4: while ($st \neq \emptyset$) {stack $st$ is not empty} do
 5:   $m \leftarrow st.top()$ {top element of stack $st$}
 6:   if ($mrk(m) == false$) {can we extend current r-path CRP with $m$} then
 7:     if (prune() == conflict) {check if taking $m$ causes conflict} then
 8:       learn() {compute reason for conflict and learn}
 9:       backtrack() {non-chronological backtracking}
10:       continue {goto while loop (line 4)}
11:     end if
12:     $mrk(m) \leftarrow true$ {extend current satisfiable r-path with $m$}
13:     $\sigma \leftarrow \sigma \cup \{Lit(m)\}$ {add $Lit(m)$ to current assignment}
14:     if ($m \in L$) {node $m$ is a leaf} then
15:       return SAT {we found a satisfiable rl-path in $G_v(\phi)$}
16:     else
17:       push all children of $m$ on $st$ {extend CRP$\langle m \rangle$ to reach a leaf}
18:     end if
19:   else
20:     backtrack () {non-chronological backtracking}
21:   end if
22: end while
23: return UNSAT {no satisfiable rl-path exists in $G_v(\phi)$}
```

The high level description of the solver is given in Algorithm 3.1. The input to the algorithm is a vpgraph $G_v(\phi) = (V, R, L, E, Lit)$ and a hpgraph $G_h(\phi) = (V', R', L', E', Lit')$ corresponding to a formula ϕ. If $G_v(\phi)$ contains a satisfiable rl-path, then the algorithm returns `SAT` as the answer. Otherwise, ϕ is unsatisfiable and the algorithm returns `UNSAT`. The algorithm uses the hpgraph $G_h(\phi)$ in various sub-routines such as `prune` and `learn`. The following data structures are used:

- *st* is a stack. It stores a subset of nodes from V that need to be explored when searching for a satisfiable rl-path in $G_v(\phi)$. Initially, the roots in $G_v(\phi)$ are pushed on the stack *st* (line 1). Let $st.top()$ return the top element of *st*. We write *st* as $[n_0, \ldots, n_k]$ where the top element is n_k and the bottom element is n_0.
- σ stores the current truth assignment as a set. Each element of σ is a literal which is true under the current assignment. For example, an assignment with sets variables a, b to true and c to false will be denoted as $\{a, b, \neg c\}$. The algorithm ensures that σ is *consistent*, that is, it does not contain contradictory literals of the form l and $\neg l$. Initially, σ is the empty set (line 2).
- *mrk* maps a node in V to a Boolean value. It identifies an r-path in $G_v(\phi)$ which is currently being considered by the algorithm to obtain a satisfiable rl-path (see Fig. 3.3(a)). We refer to this r-path as the *current r-path* (`CRP` for short). Intuitively, $mrk(n)$ is true for nodes that lie on the `CRP` ($n \in \texttt{CRP}$) and false for all other nodes in $G_v(\phi)$. More precisely, the `CRP` is obtained

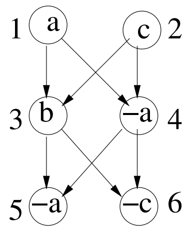


Figure 3.2: The vpgraph for formula $(a \vee c) \wedge (b \vee \neg a) \wedge (\neg a \vee \neg c)$.

by removing every node n from the stack st for which $mrk(n)$ is false. The remaining nodes constitute the `CRP`. Initially, $mrk(n)$ is set to false for every node n (line 3), thus, `CRP` is empty.

Example 3 The vpgraph for the formula $\phi = (a \vee c) \wedge (b \vee \neg a) \wedge (\neg a \vee \neg c)$ is shown in Fig. 3.2. Initially, we have st as $[2,1]$ where the top element of the stack is 1, $\sigma = \emptyset$, $mrk(n) = false$ for all $n \in \{1,2,3,4,5,6\}$. Suppose during the execution of the algorithm we have st as $[2,1,4,3,6,5]$, and $mrk(1), mrk(3)$ are $true$ and $mrk(n) = false$ for $n \in \{2,4,5,6\}$. Thus, `CRP` is $\langle 1,3 \rangle$. Observe that `CRP` is an r-path. Intuitively, the algorithm tries to extend `CRP` by one node at a time, to obtain a satisfiable rl-path. In this case `CRP` can be extended to obtain two rl-paths $\pi_1 = \langle 1,3,5 \rangle$ or $\pi_2 = \langle 1,3,6 \rangle$. However, only π_2 is satisfiable (by $\sigma = \{a, b, \neg c\}$) and is enough to show that ϕ is satisfiable.

The main part of the algorithm is the `while` loop (lines 4-22) which executes as long as st is not empty and the algorithm has not returned `SAT` on line 15. The algorithm maintains the following loop invariant.

Loop invariant: At the beginning of iteration number i of the `while` loop:

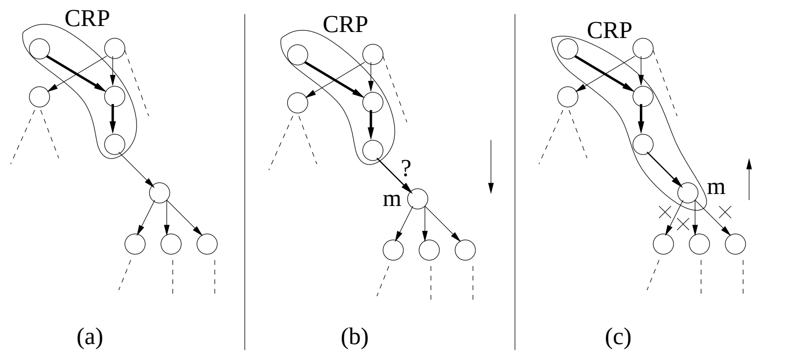


Figure 3.3: (a) Current r-path or CRP in a vpgraph (b) Can CRP be extended by node *m*? (c) Backtracking from node *m*.

let the current r-path (CRP) be $\langle n_0, \ldots, n_k \rangle$. Then the assignment σ is equal to $\{Lit(n_j) | n_j \in \texttt{CRP}\}$. That is, σ satisfies each node on CRP and thus, σ satisfies CRP. For example, suppose CRP is $\langle 1, 3 \rangle$ in the vpgraph shown in Fig. 3.2, then σ will be $\{a, b\}$.

If *st* is not empty, then the top element of the stack (denoted by *m*) is considered in line 5. There are two possibilities for node *m* according to the `if` statement in line 6.

• $mrk(m)$ is $false$: In this case the algorithm checks if the current r-path CRP can be extended by node *m* as shown in Fig. 3.3(b). This check is carried out by a call to `prune` (line 7). If `prune` returns `conflict`, then the current r-path extended by node *m* cannot lead to a satisfiable rl-path. Thus, the solver needs to backtrack from node *m*, and if possible extend CRP by some other node. This is done by calling `backtrack` on line 9 and going back to `while` loop (line 4) by using `continue` (line 10). Before backtracking a call to `learn` (line 8) is made which summarizes the reason for the conflict when CRP is extended by m. This reason is learned in form of a clause and is used later to avoid similar conflicts. We denote CRP concatenated with m as CRP$\langle m\rangle$. Depending upon the reason why there is no satisfiable rl-path with CRP$\langle m\rangle$ as prefix, the backtrack routine can pop several nodes from st (non-chronological backtracking) instead of just popping m from st.

If a call to prune results in no-conflict (line 7), then m can extend CRP. In this case execution reaches line 12. At line 12 $mrk(m)$ is set to true, which means that the new current r-path is CRP concatenated with m, that is, CRP$\langle m\rangle$. The algorithm maintains the loop invariant that the assignment σ satisfies the current r-path. In order to maintain this invariant σ now needs to satisfy node m which is on the current r-path CRP$\langle m\rangle$. This is done by adding $Lit(m)$ to σ (line 13). If m is a leaf in the vpgraph, then CRP$\langle m\rangle$ is a satisfiable rl-path. In this case SAT is returned (lines 14-15). If m is not a leaf, then the children of m are pushed on the stack (line 17). The algorithm will next attempt to extend the current r-path CRP$\langle m\rangle$.

• $mrk(m)$ is $true$: This happens when the current r-path is of the form $\langle n_0,\ldots,n_k,m\rangle$. Intuitively, the algorithm has explored all possible rl-paths with $\langle n_0,\ldots,n_k,m\rangle$ as prefix, but none of them leads to a satisfiable rl-path as shown in Fig. 3.3(c). The algorithm now backtracks from node m by calling backtrack on line 20 . Depending upon the reason why there is no satisfiable rl-path with $\langle n_0,\ldots,n_k,m\rangle$ as prefix, the algorithm can pop several nodes from st instead of just popping m.

For each node n removed from the stack during backtracking (lines 9, 20)

$mrk(n)$ is set to false again. This enables the removed nodes to be examined again on rl-paths which have not yet been explored.

We discuss the routines `prune`, `learn`, and `backtrack` in the following sections.

3.4 Search Space Pruning

This section describes the procedure `prune` called in the non-clausal SAT algorithm shown in Algorithm 3.1 (line 7). A call to `prune` checks if the current r-path `CRP` can be extended by node m or not, as shown in Fig. 3.3(b). Intuitively, `prune` returns `conflict` if there cannot be a satisfiable rl-path in vpgraph $G_v(\phi)$ with $\texttt{CRP}\langle m\rangle$ as prefix. When `prune` is called, the current r-path `CRP` is satisfied by assignment σ, which is equal to $\{Lit(n)|n \in \texttt{CRP}\}$ (maintained as a `while` loop invariant in the top level algorithm shown in Algorithm 3.1). The three cases when `conflict` is returned are as follows:

Case 1: When $\texttt{CRP}\langle m\rangle$ is not satisfiable. This happens when there is a node n on `CRP` such that $Lit(n) = \neg Lit(m)$. In this case no assignment can satisfy the r-path $\texttt{CRP}\langle m\rangle$ (Corollary 5). For example, in the vpgraph shown in Fig. 3.4(a) this conflict arises when the `CRP` is $\langle 1,3\rangle$ and m is node 5.

Otherwise, $\texttt{CRP}\langle m\rangle$ is satisfiable and $\sigma' = \sigma \cup \{Lit(m)\}$ satisfies $\texttt{CRP}\langle m\rangle$. However, it is still possible that there is no satisfiable rl-path in $G_v(\phi)$ with $\texttt{CRP}\langle m\rangle$ as prefix. These cases are described below.

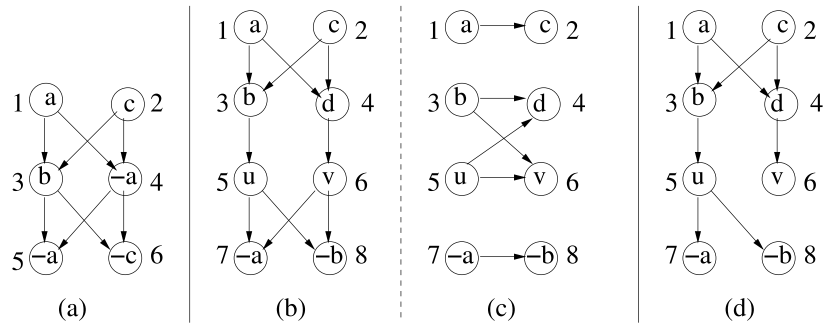


Figure 3.4: (a) Vpgraph for formula $(a \vee c) \wedge (b \vee \neg a) \wedge (\neg a \vee \neg c)$. (b,c) Vpgraph and Hpgraph for formula $(a \vee c) \wedge ((b \wedge u) \vee (d \wedge v)) \wedge (\neg a \vee \neg b)$, respectively (d) Vpgraph for formula $(a \vee c) \wedge ((b \wedge u \wedge (\neg a \vee \neg b)) \vee (d \wedge v))$.

Case 2 (Global conflict): When σ' falsifies ϕ. In this case we claim that there is no satisfiable rl-path in $G_v(\phi)$ with $\texttt{CRP}\langle m \rangle$ as a prefix. We prove this claim by contradiction. Assume that there is an rl-path π in $G_v(\phi)$ which has $\texttt{CRP}\langle m \rangle$ as prefix and is satisfiable. By definition there exists an assignment σ'' which satisfies π. From Corollary 2 we know that σ'' satisfies ϕ. In order to satisfy π, σ'' must satisfy $\texttt{CRP}\langle m \rangle$. That is, σ'' must contain $Lit(n)$ for every $n \in \texttt{CRP}\langle m \rangle$. Since $\sigma' = \{Lit(n) | n \in \texttt{CRP}\langle m \rangle\}$, it follows that $\sigma' \subseteq \sigma''$. But σ' falsifies ϕ and hence σ'' must falsify ϕ. This leads to a contradiction.

Example 4 In Fig. 3.4(b) vpgraph for formula $\phi := (a \vee c) \wedge ((b \wedge u) \vee (d \wedge v)) \wedge (\neg a \vee \neg b)$ is given. Consider the case when $\texttt{CRP}$ is $\langle 1 \rangle$ and $\sigma = \{a\}$. The algorithm checks if $\texttt{CRP}$ can be extended by node 3 ($m = 3$). Using our notation $\sigma' = \{a, b\}$. Observe that σ' falsifies ϕ by substituting $a = true, b = true$ in ϕ. There are two rl-paths $\pi_1 := \langle 1,3,5,7 \rangle, \pi_2 := \langle 1,3,5,8 \rangle$ in the vpgraph shown in Fig. 3.4(b) which have $\langle 1,3 \rangle$ as prefix. Neither of these rl-paths is satisfiable:

π_1 is not satisfiable due to conflicting nodes 1, 7 and π_2 is not satisfiable due to conflicting nodes 3, 8.

Detection of a global conflict: We use Corollary 3 to check if σ' falsifies ϕ. We check if there is an rl-path π in $G_h(\phi)$ such that σ' falsifies π. Continuing the above example, the hpgraph corresponding to ϕ is shown in Fig. 3.4(c). Observe that $\sigma' = \{a, b\}$ falsifies the rl-path $\langle 7, 8\rangle$ in Fig. 3.4(c). Thus, using Corollary 3, it follows that σ' falsifies ϕ.

If there is no global conflict, then the set of implied assignments can be found by the application of unit literal rule on $G_h(\phi)$ as described in Corollary 6.

Case 3 (Local conflict): This conflict arises when every rl-path in $G_v(\phi)$ with $\texttt{CRP}\langle m\rangle$ as prefix contains two nodes which are conflicting and one of the conflicting nodes lies on $\texttt{CRP}\langle m\rangle$. Formally, this conflict arises when for every rl-path π in $G_v(\phi)$ with $\texttt{CRP}\langle m\rangle$ as prefix there exist two nodes $k, l \in \pi$ and $k \in \texttt{CRP}\langle m\rangle$ such that $Lit(k) = \neg Lit(l)$. From Corollary 5, it follows that any rl-path π containing conflicting nodes is not satisfiable. Thus, when a local conflict occurs no rl-path in $G_v(\phi)$ with $\texttt{CRP}\langle m\rangle$ as prefix is satisfiable. Whenever there is a global conflict (case 2 above) there is also a local conflict, however, the reverse need not hold as shown by the example below.

Example 5 In Fig. 3.4(d) the vpgraph for formula $\phi := (a \vee c) \wedge ((b \wedge u \wedge (\neg a \vee \neg b)) \vee (d \wedge v))$ is shown. Consider the case when $\texttt{CRP}$ is $\langle 1\rangle$ and m is node 3 ($m = 3$). Using our earlier notation $\sigma' = \{a, b\}$. Note that σ' does not falsify ϕ, which means there is no global conflict. There are two rl-paths $\langle 1,3,5,7\rangle, \langle 1,3,5,8\rangle$ in the vpgraph shown in Fig. 3.4(d) which have $\langle 1,3\rangle$ as prefix. Both of these rl-paths contain two conflicting nodes, nodes 1,7 are conflicting on $\langle 1,3,5,7\rangle$ and nodes 3,8 are conflicting on $\langle 1,3,5,8\rangle$. Thus, there is a local conflict and the solver needs to backtrack from node $m = 3$.

Detection of global and local conflicts can be done in linear time in the size of vpgraph/hpgraph as described in the appendix B. Depending upon the type of conflict (global or local) we perform global or local learning as described below.

3.5 Learning

Learning records the cause of a conflict. This enables the preemption of similar conflicts later on in the search. In the following, a *clause* will refer to a disjunction of literals. A clause C is *conflicting* under an assignment σ iff all literals in C are falsified by σ. If a clause C is not conflicting under an assignment σ, we say C is *consistent* under σ. We distinguish between two types of learning:

Global learning: A *globally learned* clause is a clause whose consistency must be maintained irrespective of the current search state, which is given by the current r-path `CRP` (and assignment $\sigma = \{Lit(n) | n \in \texttt{CRP}\}$). That is, whenever a globally learned clause becomes conflicting under σ the solver abandons the current search state and backtracks. A globally learned clause is generated from a conflicting clause. A conflicting clause C arises in two cases as described below.

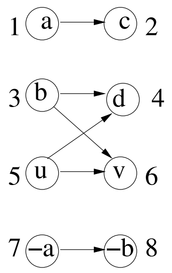


Figure 3.5: Hpgraph for formula $(a \vee c) \wedge ((b \wedge u) \vee (d \wedge v)) \wedge (\neg a \vee \neg b)$.

1. When analyzing global conflicts as described in the previous section. When a global conflict occurs there is an rl-path π in hpgraph $G_h(\phi)$ which is falsified by the assignment σ currently under consideration. The set of literals corresponding to the nodes on π gives us a clause $C := \bigvee_{n \in \pi}(Lit(n))$. Observe that C is a conflicting clause, that is, all literals occurring in C are set to false under the current assignment.

   Example 6 The hpgraph corresponding to $\phi := (a \vee c) \wedge ((b \wedge u) \vee (d \wedge v)) \wedge (\neg a \vee \neg b)$ is shown in Fig. 3.5. A global conflict occurs when the current assignment is $\sigma = \{a, b\}$, that is, σ falsifies ϕ. In this case the rl-path in the hpgraph which is falsified by σ is $\langle 7, 8 \rangle$. Thus the required conflicting clause is $\neg a \vee \neg b$.

2. When all literals of an existing globally learned clause C become false.

Once a conflicting clause C is obtained, we perform a 1-UIP (*first unique implication point*) analysis [148] to obtain a learned clause C'. Clause C' is added to the database of globally learned clauses. In order to perform 1-UIP analysis we

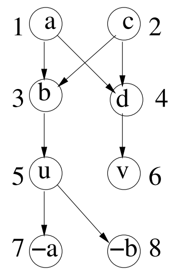


Figure 3.6: Vpgraph for formula $(a \vee c) \wedge ((b \wedge u \wedge (\neg a \vee \neg b)) \vee (d \wedge v))$.

maintain a notion of a decision level. We associate a decision level $dec(n)$ with each node n in the current r-path `CRP`. We also maintain a set of implied literals at each node (or decision level) along with the reason (set of variable assignments) which led to the implication. We follow the same algorithm as in [148] to perform the 1-UIP learning.

Local learning: A *locally learned* clause is associated to a node n in the vpgraph when a local conflict occurs at n. Suppose C is a locally learned clause at node n. Then the consistency of C needs to be maintained only when n is part of the current search state, that is, $n \in$ `CRP`. If n does not lie on `CRP`, then the consistency of C is irrelevant. This is in contrast to a globally learned clause whose consistency must always be maintained.

Example 7 Consider the local conflict which occurs in the vpgraph in Fig. 3.6 when `CRP` is $\langle 1 \rangle$ and it is checked if `CRP` can be extended by $m = 3$. In this case every rl-path in vpgraph with $\langle 1,3 \rangle$ as prefix contains two conflicting nodes one of which lies on $\langle 1,3 \rangle$. The rl-path $\langle 1,3,5,7 \rangle$ has conflicting nodes 1,7 and the rl-path $\langle 1,3,5,8\rangle$ has conflicting nodes 3,8. In this case a clause $Lit(7) \vee Lit(8) = \neg a \vee \neg b$ can be learned at node 3. Intuitively, when we consider extending the `CRP` with node m the (locally) learned clauses at node m must be consistent with the assignment $\sigma = \{Lit(n) | n \in \texttt{CRP}\langle m\rangle\}$. Otherwise, a local conflict will occur at m causing the solver to backtrack. Having learned clauses at node m avoids repeating the work done in detecting the same local conflict. For the vpgraph in Fig. 3.6, when `CRP` is $\langle 2\rangle$ and $m = 3$, $\sigma = \{c, b\}$ is consistent with the learned clause $\neg a \vee \neg b$ at node 3, thus, the solver cannot get the same local conflict at node 3 as before (when `CRP` was $\langle 1\rangle$ and $m = 3$).

If a local conflict occurs when extending `CRP` by node m, then a clause is learned at node m as follows: For each rl-path π having $\texttt{CRP}\langle m\rangle$ as prefix let $\omega_1(\pi), \omega_2(\pi)$ denote the pair of conflicting nodes on π. Without loss of generality assume that $\omega_1(\pi)$ lies on $\texttt{CRP}\langle m\rangle$. Then the learned clause C at node m is given by $\bigvee_\pi Lit(\omega_2(\pi))$. Consistency of C must be maintained only when considering rl-paths passing through m.

3.6 Non-chronological Backtracking

Analyzing conflicts to determine their causes enables modern SAT solvers to backtrack *non-chronologically* to earlier levels in the search tree, potentially pruning large portions of the search space. This technique is also applicable in our SAT procedure. The backtracking routine called in our SAT procedure depends on the type of conflict that invoked the backtracking procedure:

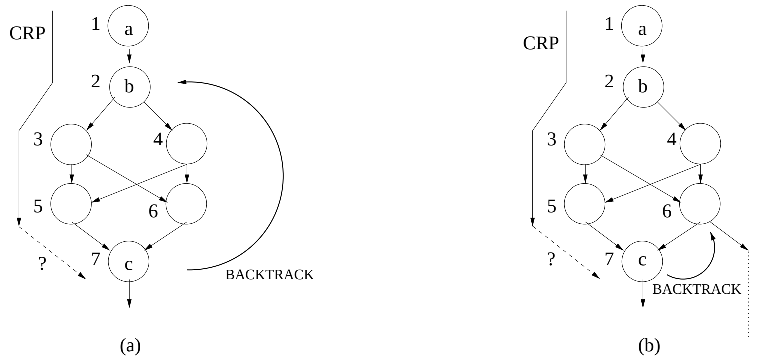


Figure 3.7: Non-chronological backtracking example.

Non-chronological backtracking on a global conflict: When a global conflict occurs the solver calls a backtracking procedure similar to that in CNF SAT solvers. Suppose a global conflict occurs when the solver attempts to extend the `CRP` with node *m*. In this case the learning procedure produces an *asserting clause* [148] C. That is, only one literal in C called the *asserting literal* $al(C)$ is assigned at the current decision level (corresponding to node *m*), and remaining literals were assigned at earlier decision levels (corresponding to nodes on `CRP`). The solver identifies the highest decision level $maxd(C)$ among the literals of $C \setminus \{al(C)\}$. The solver backtracks to the node m' corresponding to the decision level $maxd(C)$. At m' the clause C becomes a unit clause and $al(C)$ is set to true. The search proceeds from m' onwards.

Non-chronological backtracking on a local conflict: When a local conflict occurs the solver backtracks non-chronologically by analyzing the structure of the vpgraph and the conflict clause produced due to local conflict.

Example 8 Consider the vpgraph shown in Fig. 3.7(a). Let CRP be $\langle 1,2,3,5 \rangle$ and the solver examines if CRP can be extended by node 7. Suppose a local conflict occurs at node 7 and a clause $\neg a \vee \neg b \vee \neg c$ is learned at node 7. Assume that all the paths starting from nodes 2, 3, 4, 5, 6 pass through node 7. This conflict can be resolved only if we backtrack to the nodes containing literal a or literal b (assuming c was assigned at node 7). Since node 2 comes later on CRP, our algorithm backtracks to node 2. Note that backtracking prevents us from examining rl-paths such as $\langle 1,2,3,6,7,\ldots \rangle, \langle 1,2,4,5,7,\ldots \rangle, \langle 1,2,4,6,7,\ldots \rangle$ all of which would lead to the same conflict at node 7. The suffix of CRP starting from node 2 onwards is removed. The search procedure attempts to extend CRP $\langle 1 \rangle$ with other unexamined successors of node 1. In this case there is no other unexamined successor of node 1, so the solver backtracks from node 1 to return unsatisfiable answer.

Now consider a variation of Fig. 3.7(a) in Fig. 3.7(b). Node 6 has an outgoing path that does not pass through node 7. As before, a local conflict occurs at node 7. However, now instead of backtracking to node 2 our algorithm backtracks to node 6. This is because there are alternative rl-paths such as $\langle 1,2,3,6,\ldots \rangle, \langle 1,2,4,6,\ldots \rangle$ through node 6 which could be satisfiable.

3.7 Decision Heuristics

In modern DPLL-based SAT solvers *decision heuristics* play an important role in pruning the search space by identifying the variables to be assigned next. In our algorithm decision heuristics are used to decide the order in which the children of the last node on the CRP will be examined. More precisely, we use decision heuristics when pushing the children of m on the stack (Algorithm 3.1, line 17). The children near the end of stack get examined before the other children on the stack.

Some of our decision heuristics make use of *literal activity*. The activity of a literal indicates its usefulness (participation) in conflicts so far. It is updated in a similar manner as in zChaff [117]. The activity n of a node in vpgraph is simply the activity of literal $Lit(n)$. We describe a few decision heuristics used when pushing children of m (line 17) below.

1. Push the children in the order they occur in the adjacency list of m.
2. Push the children of m in a random order.
3. Push the children of m in the ascending order of activity. The higher activity children will be examined before other children.
4. Divide the children of m in two sets S_1 and S_2. Each node $n \in S_1$ has $Lit(n)$ already set to true. S_2 contains the remaining children of m. Push nodes in S_2 in the stack followed by nodes in S_1. Intuitively, the nodes in S_1 are *satisfied* and `prune` will not return a `conflict` for a node in S_1. Thus, a satisfying assignment or a conflict will be reached quickly by examining the nodes in S_1 before the nodes in S_2.
5. Form the sets S_1 and S_2 as above. Push the nodes in S_2 in ascending order

Bench -mark	#Probs	SatMate		MiniSat		BerkMin		Siege		zChaff	
		Time	Sol	Time	Sol	Time	Sol	Time	Sol	Time	Sol
QG6	256	23266	235	49386	179	46625	184	46525	184	47321	180
QG6*	256	23266	235	37562	211	15975	239	30254	225	45557	186
Mboard	19	4316	12	4331	12	4947	11	4505	12	5029	11
Pigeon	19	5110	11	6114	9	5459	10	6174	9	5483	11

Table 3.1: Comparison between SatMate, MiniSat, BerkMin, Siege, zChaff. ”Time” gives total time in seconds and ”Sol” gives #problems solved within time-out of 600 seconds/problem.

of their activity. Then push nodes in S_1 in ascending order of activity.

In our experiments heuristic five outperforms the other decision heuristics.

3.8 Experimental Results

The experiments were performed on a 1.5 GHZ AMD machine with 3 GB of memory running Linux. The techniques described in the chapter have been implemented in a SAT solver called SatMate [17]. The non-clausal input formula is given in `EDIMACS` [6] or `ISCAS` format. SatMate also accepts CNF inputs in `DIMACS` format. We compare SatMate against four CNF SAT solvers MiniSat version 1.14 [8], BerkMin version 561 [84], Siege version 4 [18], and zChaff version 2004.5.13 [117][1].

QG6 benchmarks The authors of [116] provided us with a benchmark set called QG6 which consists of 256 non-clausal formulas of varying difficulty.

[1]These experiments were carried out in early 2006.

These benchmarks were generated during the construction of classification theorems for quasigroups [116]. The CNF version of these problems was also made available to us by the authors of [116]. The CNF version was obtained by directly expressing the problem of classifying quasigroups into CNF as opposed to the translation of non-clausal formulas into CNF. The non-clausal versions of these benchmarks have 300 variables and 7500 gates (AND, OR gates) on average, while the CNF versions have 1700 variables and 7500 clauses on average. We ran SatMate on the non-clausal formulas and CNF SAT solvers on the corresponding CNF formulas from QG6 suite.

QG6* benchmarks We translated the non-clausal formulas from the QG6 suite into CNF by introducing new variables [124]. The CNF formulas obtained after translation have 7500 variables and 30000 clauses on average. We ran CNF SAT solvers on the CNF formulas obtained after translation. Note that we still ran SatMate on the non-clausal formulas.

Mboard benchmarks encode the mutilated-checkerboard problem.

Pigeon benchmarks encode the pigeon hole principle with n holes and $n+1$ pigeons.

Both QG6 and QG6* benchmarks contain a mixture of satisfiable and unsatisfiable problems. All problems in the Mboard and Pigeon benchmarks are unsatisfiable.

The experimental results are summarized in Table 3.1. The column ”#Probs” gives the number of problems in each benchmark set. There was a timeout of 10 minutes per problem per solver. For each solver we report two quantities: 1) ”Time” is the total time spent in seconds when solving problems in a given benchmark, including the time spent (= timeout) for each instance not solved within timeout. 2) ”Sol” gives the total number of problems that were solved within timeout.

Summary of results in Table 3.1: On QG6 benchmarks SatMate solves around 50 more problems and it is approximately 2 times faster than the CNF SAT solvers MiniSat, BerkMin, Siege, and zChaff. On QG6* benchmarks SatMate performs better than MiniSat, zChaff, Siege. However, BerkMin outperforms SatMate on QG6* benchmarks. The difference in the performance of CNF SAT solvers on QG6 and QG6* benchmarks shows how the differences in the encoding of a given problem to CNF can significantly impact the performance of CNF SAT solvers. The performance of SatMate on Mboard and Pigeon benchmarks is slightly better than the CNF SAT solvers.

Table 3.2 summarizes the performance of SatMate and four CNF SAT solvers on various individual problems. Problems `dnd02, brn13, icl39, icl45` are from QG6 benchmark suite. Problems `q2.14,cache.inv12` are generated by UCLID verification tool [22]. The sub-column ”Time” gives the time required for SAT solving (in seconds). For SatMate we report the number of local conflicts and the number of global conflicts (Section 3.4) in the ”Local confs” and ”Global confs” sub-columns, respectively. A timeout of 1 hour was set per problem. We

Problem	SatMate			MiniSat	BerkMin	Siege	zChaff
	Time	Local confs	Global confs	Time	Time	Time	Time
dnd02	**174**	23500	15588	1308	1085	1238	TO
brn13	**181**	20699	20062	1441	1673	1508	TO
icl39	**200**	22683	14069	TO	TO	2629	TO
icl45	TO	4850	72106	TO	2320	**1641**	TO
q2.14	237	113	15863	**23**	24	34	88
cache.inv12	58	659	7131	**1**	**1**	**1**	2

Table 3.2: Comparison on individual benchmarks. Timeout is 1 hour per problem per solver. ”Time” sub-column gives time taken in seconds.

denote timeout by ”TO”. In case of timeout we report the number of conflicts just before the timeout for SatMate.

Performance of SatMate is correlated with the number of local conflicts and global conflicts. A local conflict is a conflict that occurs in a part of a formula and it depends on the structure of the vpgraph. There is no equivalent of local conflict in CNF SAT solvers. In CNF SAT solvers a conflict arises when the current assignment falsifies an original/learned clause which is equivalent to a global conflict. As shown in Table 3.2 the number of local conflicts is usually comparable to the number of global conflicts on the benchmarks where SatMate outperforms CNF SAT solvers. Indeed the performance of SatMate degrades if no local conflict detection and local learning is done.

On SAT problems arising from verification applications such as bounded model checking the performance of SatMate is usually worse than the SAT solvers based on DPLL. Intuitively, this happens because there is a lot of “irrelevant” information in the verification benchmarks, that is, only a small fraction of the variables are important for (un)satisfiability of the given formula. The decision heuristics in the DPLL SAT solvers are able to quickly identify and branch on the important variables. The General Matings solver is constrained to follow the vpgraph structure and does not have much flexibility in terms of the variables to branch on. This drawback can be addressed by dividing the vpgraph into a number of smaller vpgraph components and searching for satisfiable rl-paths in each component in some order. Decision heuristics can be used to select the vpgraph component to examine before other vpgraph components.

3.9 Chapter Summary

We presented a new non-clausal SAT solver based on the General Matings approach. This approach involves the search for a vertical path which does not contain opposite literals in the vertical-horizontal path form (vhpform) of a given negation normal form formula. The main challenge in obtaining an efficient SAT solver based on the General Matings approach is to prevent the enumeration of vertical paths. We presented new techniques for preventing the enumeration of vertical paths. Experimental results show that on certain classes of non-clausal benchmarks our SAT solver has a performance comparable or better than the current CNF SAT solvers. Overall, our results show the promise of the General Matings approach in building SAT solvers.

Chapter 4

DPLL based SAT Solver

In this chapter we present a SAT solver that checks the satisfiability of a NNF formula ϕ by applying the DPLL algorithm [70, 71] to the hpgraph of ϕ. Our solver also utilizes the vpgraph of ϕ in certain steps of SAT solving. If the input formula is not in NNF it can be converted to an equi-satisfiable NNF formula by using the techniques discussed in chapter 2.

4.1 Top Level DPLL Algorithm

The high level organization of our DPLL based SAT solver is shown in Algorithm 4.1. The input to the algorithm is the hpgraph and vpgraph of a formula ϕ. The output is `SAT` if ϕ is satisfiable and `UNSAT` if ϕ is unsatisfiable. The top level algorithm is similar to other state-of-the-art DPLL based SAT solvers.

The main body of the algorithm consists of a while loop which executes as

Algorithm 4.1 Top Level Routine in DPLL Based SAT Solver

Input: Hpgraph $G_h(\phi)$ and vpgraph $G_v(\phi)$

Output: Return `SAT` if the formula is satisfiable, else return `UNSAT`

```
 1: while (true) do
 2:   if (decide_next_branch()) then
 3:     while (bcp() == conflict) do
 4:       blevel = analyze_conflict()
 5:       if (blevel == 0) then
 6:         return UNSAT
 7:       else
 8:         backtrack(blevel)
 9:       end if
10:     end while
11:   else
12:     return SAT
13:   end if
14: end while
```

long as `SAT` or `UNSAT` is not returned. In each iteration of the outer while loop we first call `decide_next_branch()` in order to identify an unassigned variable. If an unassigned variable is found it is assigned a truth value. After assigning a new variable Boolean constraint propagation is performed by calling the `bcp()` routine. If the current truth assignment falsifies the formula the `bcp()` routine returns a `conflict`. In case of a conflict the `analyze_conflict()` routine is called in order to perform learning. The `analyze_conflict()` routine also identifies a backtracking level `blevel` where the solver needs to backtrack to in order to avoid a similar conflict. If the backtracking level is zero, then it means that the formula is unsatisfiable and `UNSAT` is returned. The `backtrack(blevel)` call performs the task of *non-chronological backtracking* by erasing all the assignments between the current decision level and the `blevel`. After backtracking to `blevel` the `bcp()` routine is called again. This is because at least one new variable gets assigned after backtracking and this can lead to more variable assignments or a conflict.

If `decide_next_branch()` returns false, it means that all variables have been assigned. In this case the algorithm returns `SAT`.

A main difference between the existing DPLL SAT solvers and our solver is in the `bcp()` routine. In our solver the BCP algorithm uses the hpgraph and the vpgraph of a given formula. The focus of this chapter is to explain the `bcp()` routine in detail.

4.2 Contributions

Our contributions can be summarized as follows:

- The most crucial component of our DPLL SAT solver is an efficient Boolean Constraint Propagation (BCP) algorithm on the hpgraph. Let V denote the set of variables in ϕ. Given an assignment σ of truth values to a set of variables $W \subseteq V$, the BCP algorithm determines if σ falsifies ϕ, else it provides the set of implied assignments (unit literals). We describe an algorithm for performing BCP on hpgraph that *generalizes* the *two-watched literal scheme* [117] found in CNF SAT solvers.

  In particular, a "watch" in an hpgraph corresponds to a *node cut* in the hpgraph. By maintaining two node cuts for each connected component in the hpgraph we achieve the same effect as the two watched-literal scheme found

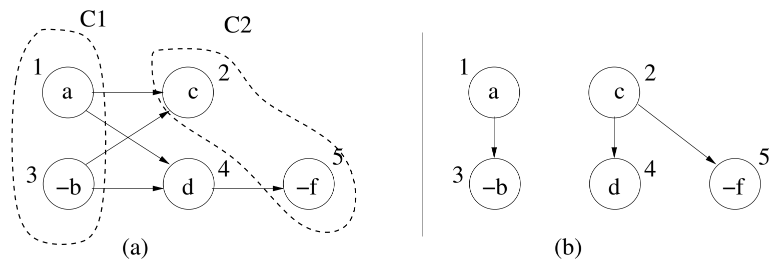


Figure 4.1: Let ϕ be $(a \wedge \neg b) \vee (c \wedge (d \vee \neg f))$. (a) The hpgraph of ϕ. Two node cuts C_1, C_2 are shown. (b) The vpgraph of ϕ.

in the CNF SAT solvers. Fig. 4.1(a) shows two node cuts C_1, C_2 (possible watches) for a hpgraph. Two node cuts allow watching two nodes (literals) on each path (clause) in a hpgraph component. The two-watched literal scheme used in CNF SAT solvers is a special case of our algorithm (when hpgraph represents a CNF formula). As in CNF SAT solvers non-chronological backtracking is cheap as the node cuts are not updated when backtracking.

- We show how to update the node cuts (watches) in the hpgraph efficiently by using the vpgraph of the given formula. We show that a *minimal* cut in a hpgraph corresponds to a path in the corresponding *vpgraph*. Thus, finding a small node cut in a hpgraph corresponds to finding a path in the corresponding vpgraph. For example, notice that paths $\langle 1,3 \rangle, \langle 2,5 \rangle$ in the vpgraph shown in Fig 4.1(b) correspond to cuts C_1, C_2, respectively, in the hpgraph shown in Fig 4.1(a).

- We have carefully implemented these ideas in a non-clausal SAT solver called `NFLSAT` (Non-clausal FormuLas SATisfiability checker). We evaluate the solver on a collection of 2541 non-clausal industrial benchmarks obtained from publicly available sources. Our solver outperforms the top three CNF SAT solvers of SAT 2007 competition (industrial category) in terms of number of problems solved and runtime. NFLSAT is also competitive with the winners of SAT-Race 2008.

4.3 Preliminaries

Definition 7 *Given an assignment σ to a subset of variables in ϕ, we say that there is* **conflict** *iff σ falsifies ϕ.*

Definition 8 *Given an assignment σ to a subset of variables in ϕ, we say that a literal l is an* **implied (unit)** *iff l must be set to true in order to obtain a satisfying assignment.*

We use the hpgraph of ϕ in order to detect conflicts. We say an assignment *falsifies* a node n in $G(\phi)$ iff the assignment falsifies $Lit(n)$. The following corollary adapts the Theorem 2 (Section 2.2) to the hpgraph.

Corollary 7 *Given an assignment σ to variables in ϕ the following are equivalent:*

1. σ falsifies ϕ

2. there exists a rl-path π in $G_h(\phi)$ such that σ falsifies every node on π

3. there is a conflict due to σ

Example 9 Consider the hpgraph for a formula ϕ in Fig. 4.2. The assignment $\sigma := \{p, q\}$ falsifies every node on rl-path $\langle 4, 5 \rangle$. Thus, σ falsifies ϕ.

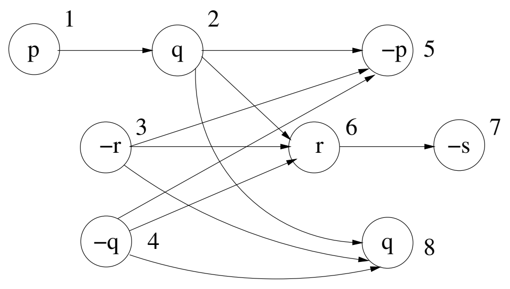


Figure 4.2: Hpgraph for a formula.

We use the hpgraph of ϕ in order to detect implied literals due to σ.

Definition 9 *Let σ be an assignment to variables in ϕ. If there is a rl-path π in $G_h(\phi)$ and a node $m \in \pi$ such that σ falsifies every node $n \in \pi, n \neq m$ and $Lit(m)$ is not assigned in σ, then we say that $Lit(m)$ an* **h-implied literal** *and m is an* **h-implied node**.

The following corollary states that an h-implied literal is also an implied literal.

Corollary 8 *Given an assignment σ to variables in ϕ. If a literal l is an h-implied literal in $G_h(\phi)$, then l is an implied literal.*

Proof. We show that $\sigma \cup \{\neg l\}$ will falsify ϕ. Since l is h-implied there is a rl-path π in $G_h(\phi)$ and a node $m \in \pi$ such that σ falsifies every node $n \in \pi, n \neq m$ and $Lit(m) = l$. Observe that $\sigma \cup \{\neg l\}$ falsifies every node on π. Thus, by Corollary 7 $\sigma \cup \{\neg l\}$ falsifies ϕ. Therefore, in order to obtain a satisfying assignment l must be set to true given σ as the current assignment. $\square$

Example 10 Consider the hpgraph shown in Fig. 4.2 and an assignment $\sigma = \{\neg p, \neg q, s\}$. σ falsifies all but node 6 on the rl-path $\langle 1,2,6,7\rangle$ in the hpgraph. It follows from Corollary 8 that $Lit(6) = r$ is an implied literal.

The use of Corollary 8 to detect implied literals is not complete, that is, certain implied literals may not be h-implied literals. Consider $\sigma = \{\neg p\}$ and the rl-path $\pi := \langle 1,2,8\rangle$ in Fig. 4.2. π corresponds to a clause $Lit(1) \vee Lit(2) \vee Lit(8) = p \vee q \vee q = p \vee q$. Since p is false under σ, q must be set to true in order to satisfy the clause $p \vee q$. However, according to definition 9, q is not an h-implied literal because of the multiple occurrences of q in nodes 2,8. Thus, q will not be detected as an implied literal in our SAT algorithm. In practice, the number of implied literals that are not h-implied are quite less and failure to detect such implied literals does not lead to worse performance. We experimented with a more complicated algorithm that detects an implied literal l even if l occurs multiple times on a rl-path. However, there was no performance improvement as compared to an algorithm using Corollary 8 to detect implied literals.

4.4 Boolean Constraint Propagation on the Hpgraph

Let V denote the set of variables in a given formula ϕ. Given an assignment σ of truth values to a set of variables $W \subseteq V$, the Boolean constraint propagation (BCP) algorithm detects two cases. (1) It reports if σ falsifies ϕ (conflict). (b) If there is no conflict, the BCP algorithm provides a set of implied (unit) literals.

Before we describe the BCP algorithm on a hpgraph, we briefly review the

BCP algorithm used in modern CNF SAT solvers.

4.4.1 Review of BCP in CNF SAT solvers

Most modern CNF SAT solvers use the *two-watched literal scheme* [117] in order to obtain an efficient BCP algorithm. Suppose we are given a CNF formula ϕ. Let C be a clause in ϕ. We assume C has at least two distinct literals [1]. In the two-watched literal scheme two *watches* are associated with C. A *watch* is simply a literal l occurring in C. Before the search (DPLL algorithm) starts any two literals in C can be designated as its watches.

Let l_1, l_2 be the watches corresponding to C. Four cases arise depending upon the status of l_1, l_2 given the current assignment σ.

Case A: Both l_1, l_2 are not false. In this case there cannot be any conflict or an implied literal due to C. The clause C is not even examined during BCP. This case occurs most often in practice and the use of watches enable efficient handling of this case.

The clause C is examined only when one of its watches becomes false. Without loss of generality assume that l_2 becomes false in the remaining three cases.

Case B: If l_1 is already true, then C is already satisfied. In this case nothing needs to be done even though l_2 is false.

Otherwise, the solver tries to replace the falsified watch (l_2) by another watch that is not false. If there is a literal l_3 in C that is not false, then l_2 is replaced by

[1] Clauses of length one are treated specially, the literal in such a clause is assigned at the earliest decision level itself.

l_3. However, such a literal (l_3) may not exist in the remaining two cases:

Case C (conflict): All the literals in C are false. In this case C is false under the current assignment.

Case D (implied literal): l_1 is unassigned but all other literals in C are false. In this case, l_1 is reported as an implied literal.

The main benefit of the two-watched literal scheme is that it reduces the number of times the solver examines the clauses in a given CNF formula. This is crucial for obtaining efficient solvers that can handle CNF formulas with millions of clauses. Another advantage is that the *non-chronological backtracking* is cheap. This is because the watched literals do not need to be updated during backtracking.

We now describe how the two-watched literal scheme found in CNF SAT solvers can be *generalized* to obtain an efficient algorithm for BCP on a hpgraph. It will be seen that the two-watched literal scheme used in the CNF solvers is a particular instance of our algorithm.

4.4.2 Generalizing Two-watched Literal Scheme to Two-watched Cut Scheme for Hpgraph

Let ϕ be a NNF formula. We are given an assignment σ to a subset of variables occurring in ϕ. The BCP algorithm uses the hpgraph $G_h(\phi)$ of ϕ.

Definition 10 *Given $G = (V, E, R, L, Lit)$ we say that $C \subseteq V$ is a* **rl-cut** *in G iff removal of all nodes in C from G disconnects all rl-paths in G.*

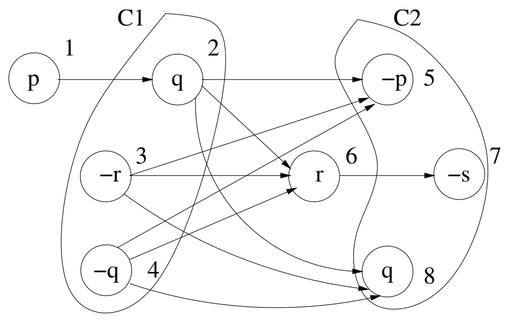


Figure 4.3: An hpgraph. Two rl-cuts $C_1 := \{2,3,4\}, C_2 := \{5,7,8\}$ are shown.

For example, $\{1,3,4\}, \{2,3,4\}, \{5,6,8\}, \{5,7,8\}$ are some of the rl-cuts in the hpgraph shown in Fig. 4.3. The node set $\{2,7,8\}$ is not an rl-cut as it does not disconnect the rl-paths $\langle 3,5\rangle, \langle 4,5\rangle$.

The following corollary states that an rl-cut contains at least one node from each rl-path.

Corollary 9 *Let C be a rl-cut in $G_h(\phi)$. For every rl-path π in $G_h(\phi)$ there exists a node n such that $n \in \pi$ and $n \in C$.*

Definition 11 *Two rl-cuts C_1, C_2 are said to be* **node-disjoint** *if $C_1 \cap C_2 = \emptyset$.*

For example the rl-cuts $\{2,3,4\}, \{5,7,8\}$ in Fig. 4.3 are node-disjoint.

Watches in a Hpgraph Each rl-path in a hpgraph corresponds to a clause. Let the clause corresponding to an rl-path π be C. In order to apply the two-watched literal scheme found in CNF SAT solvers we want to *watch* two nodes n_1, n_2 on π. This in turn amounts to watching two literals $Lit(n_1), Lit(n_2)$ in C. However, there are usually exponentially many paths (clauses) in a hpgraph. So it is expensive to maintain watches for each rl-path (clause) explicitly.

This intuition leads us to define a *watch* in a hpgraph as a *rl-cut* in the hpgraph. By taking a rl-cut as a watch we make sure that at least one node on every rl-path is present in our watch (Corollary 9). This in turn corresponds to watching a literal on each clause in the hpgraph.

Example 11 The rl-cut $C := \{2,3,4\}$ is a possible watch for the hpgraph in Fig. 4.3. Note that C contains at least one node from each rl-path in Fig. 4.3. Watching node 3 on rl-paths $\langle 3,5\rangle, \langle 3,6,7\rangle, \langle 3,8\rangle$, amounts to watching literal $Lit(3) = \neg r$ on clauses $\neg r \vee \neg p, \neg r \vee r \vee \neg s, \neg r \vee q$, respectively.

In order to get the effect of the two-watched literal scheme we watch two rl-cuts in the hpgraph. By maintaining two rl-cuts for a hpgraph we are able to watch two nodes (literals) on each rl-path (clause) in the hpgraph.

Example 12 The rl-cuts $C_1 := \{2,3,4\}$ and $C_2 := \{5,7,8\}$ are two possible watched cuts for the hpgraph in Fig. 4.3. For the rl-path $\langle 1,2,6,7\rangle$, the rl-cuts C_1, C_2 allow us to watch nodes 2,7.

Definition 12 *Given $G = (V,E,R,L,Lit)$, a partial assignment σ, and a rl-cut $W \subseteq V$. We say that W is* **acceptable** *iff there is no node $m \in W$ such that $Lit(m)$ is false in σ. We say that W is* **satisfied** *iff for all $m \in W$ $Lit(m)$ is true in σ.*

For example, given the hpgraph in Fig. 4.3 and $\sigma = \{q\}$ the rl-cuts $\{5,6,8\}, \{5,7,8\}$ are acceptable. The rl-cuts $\{2,3,4\}, \{1,3,4\}$ are not acceptable given σ.

4.4.3 High Level Description of BCP on Hpgraph Using the Two-watched Cuts

For a given hpgraph $G_h(\phi) = (V, E, R, L, Lit)$ we maintain two rl-cuts C_1, C_2 (watches). Before the DPLL algorithm starts C_1, C_2 can be initialized to any two rl-cuts in $G_h(\phi)$ which are node-disjoint[2]. As the search progress the algorithm tries to maintain the invariant that at least one of C_1, C_2 is acceptable. The algorithm also tries to maintain C_1, C_2 as node-disjoint as possible. This is useful for detecting implied literals.

We intuitively describe the various cases that may arise during the BCP on a hpgraph below. In the next section we formalize these cases. Each of the cases below generalize the cases that occur in the two-watched literal scheme for CNF SAT solvers.

Case A: Both rl-cuts (watches) C_1, C_2 are node-disjoint and acceptable. Then there can be no conflict or h-implied literals due to the current assignment. This is because each clause in the hpgraph contains two literals that are not false. In this case there is no need to look at any other part of the hpgraph.

Example 13 In Fig. 4.3 let $C_1 := \{2,3,4\}, C_2 := \{5,7,8\}$ and $\sigma = \{\neg p\}$. Observe that both C_1, C_2 are acceptable rl-cuts and are node-disjoint. It can be seen that there is no conflict or h-implied literals in the hpgraph.

Suppose one of the rl-cuts say C_2 is no longer acceptable. Then we have the following cases.

[2]This can always be ensured as explained in the next section.

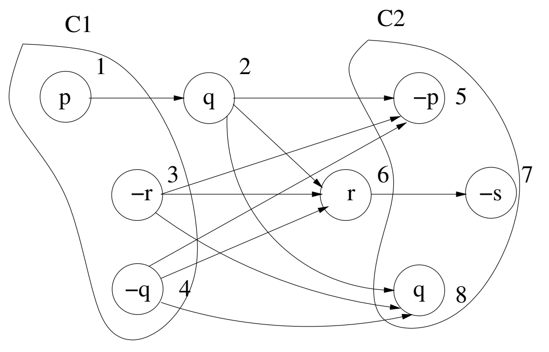


Figure 4.4: An hpgraph. Two rl-cuts $C_1 := \{1,3,4\}, C_2 := \{5,7,8\}$ are shown.

Case B: For each node $n \in C_1$, $Lit(n)$ is already true, that is, C_1 is *satisfied.* In this case there cannot be any conflict or an h-implied literal in the hpgraph. Intuitively, every clause present in the hpgraph is satisfied. The algorithm leaves C_2 unchanged in this case.

Example 14 In Fig. 4.4 let $C_1 := \{1,3,4\}, C_2 := \{5,7,8\}$ and $\sigma = \{p, \neg r, \neg q\}$. Observe that C_2 is not acceptable as $Lit(5), Lit(8)$ are false under σ. However, C_1 is satisfied. In this case we do not update C_2.

If the previous cases do not apply the algorithm tries to find a replacement rl-cut for C_2. When searching for a replacement to C_2 the algorithm tries to find a rl-cut that is as different from C_1 as possible. Intuitively, this is similar to why we keep the two-watched literals in a clause as distinct in the CNF two-watched literal scheme. If a replacement cut C_3 is found such that C_3 is acceptable and $C_3 \cap C_1 = \emptyset$, then C_2 is replaced by C_3. Otherwise, we have the following two cases.

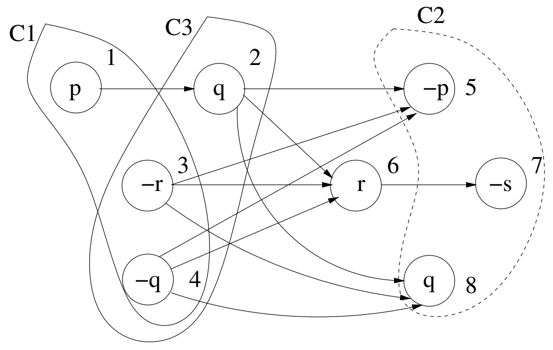


Figure 4.5: An hpgraph. Three rl-cuts $C_1 := \{1,3,4\}, C_2 := \{5,7,8\}, C_3 := \{2,3,4\}$ are shown.

Case C (conflict): If there is no acceptable rl-cut in the hpgraph. In this case the current assignment σ falsifies the given formula.

Example 15 In Fig. 4.4 let $C_1 := \{1,3,4\}, C_2 := \{5,7,8\}$ and $\sigma = \{p,r\}$. In this case both C_1, C_2 are not acceptable and there is no possible replacement for them. This is expected as the clause $(\neg r \vee \neg p)$ corresponding to the rl-path $\langle 3,5 \rangle$ is false.

Case D (implications): If there an acceptable cut C_3 such that C_3 is acceptable but $C_3 \cap C_1 \neq \emptyset$. In this case, for every $n \in C_1 \cap C_3$ the corresponding literal $Lit(n)$ is an h-implied literal (assuming $Lit(n)$ is not already true). If $C_3 \neq C_1$, then C_2 is replaced by rl-cut C_3.

Example 16 In Fig. 4.5 let $C_1 := \{1,3,4\}, C_2 := \{5,7,8\}$ and $\sigma = \{p\}$. Observe that C_2 is not acceptable as $Lit(5)$ is false under σ. Also note that case B does not hold as C_1 is not satisfied. Thus, we seek a replacement for C_2. Note that any new acceptable cut must include nodes 3,4 since as they are the only possible nodes that can be watched on the paths $\langle 3,5\rangle, \langle 4,5\rangle$, respectively. Thus, a possible rl-cut C_3 is $\{2,3,4\}$. Both C_1, C_3 contain nodes 3,4. It can be seen that $Lit(3), Lit(4)$ are precisely the h-implied literals. $Lit(3) = \neg r$ is h-implied due to the rl-path $\langle 3,5\rangle$, which corresponds to the clause $\neg r \vee \neg p$. Similarly, $Lit(4) = \neg q$ is h-implied due to the rl-path $\langle 4,5\rangle$, which corresponds to the clause $\neg q \vee \neg p$. Since h-implied literals are also implied literals it follows that $\neg r, \neg q$ are implied literals given the current assignment.

4.5 Formalizing the Two-watched Cut Scheme for BCP on Hpgraph

We define the length of a rl-path as the number of nodes on the rl-path. Let $\pi = \langle n_0 \rangle$ be a rl-path in $G_h(\phi)$ of length one. The rl-path π corresponds to a unit clause $Lit(n_0)$. Our algorithm removes π from $G_h(\phi)$ and sets $Lit(n_0)$ to *true* during the pre-processing phase itself. This is done for all rl-paths that have length one. In the following we assume that each rl-path in $G_h(\phi)$ has a length greater than one. This ensures that there are always two node-disjoint rl-cuts in the given hpgraph.

Recall that there is *conflict* in $G_h(\phi)$ due to an assignment σ iff σ falsifies ϕ iff there is a rl-path π in $G_h(\phi)$ such that $Lit(m)$ is false for every $m \in \pi$.

Definition 13 *Given a hpgraph $G_h(\phi) = (V, E, R, L, Lit)$ and an assignment σ. We say that a node $n \in V$ is* **p-assigned (possibly assigned)** *if the following condi-*

tions hold:

(a) there is a rl-path π in $G_h(\phi)$ and n lies on π, and

(b) for every node $m \neq n$ and $m \in \pi$, $Lit(m)$ is false under σ

If n is a p-assigned node, then $Lit(n)$ is said to be a p-assigned literal.

Let n be a p-assigned node. Depending upon the status of $Lit(n)$ under the current assignment σ we have three cases:

1. $Lit(n)$ is unassigned under σ. In this case observe that $Lit(n)$ is an h-implied literal and n is an h-implied node. Also note that $Lit(n)$ is an implied literal.

2. $Lit(n)$ is false. In this case there exists a rl-path π such that $n \in \pi$ and every node on π is falsified. This corresponds to a case when we have a conflict.

3. $Lit(n)$ is true. In this case there exists a rl-path π such that $n \in \pi$ and every node $m \in \pi, m \neq n$ is falsified. Since $Lit(n)$ is true the clause corresponding to π is satisfied.

Corollary 10 *Let n be a node in hpgraph. If n is h-implied, then n is also p-assigned. Thus, if there is no p-assigned node in the hpgraph, then there cannot be any h-implied node.*

Corollary 11 *If every node on a rl-path π in $G_h(\phi)$ is falsified by an assignment σ (conflict case), then every node on π is p-assigned. Thus, if there is no p-assigned node in the hpgraph given an assignment σ, then there cannot be a conflict due to σ, that is, σ does not falsify ϕ.*

The following theorem formalizes the **case C** in Section 4.4.3. It states that there is no conflict due to the current assignment if and only if we can find an acceptable rl-cut in the hpgraph.

Theorem 4 *Given hpgraph $G_h(\phi)$, an assignment σ. The following are equivalent:*
(a) There is no conflict in $G_h(\phi)$ due to σ.
(b) There exists a rl-cut C in $G_h(\phi)$ such that C is acceptable.

Proof. (a) $\Rightarrow$ (b): Define C to be the collection of all nodes n in $G_h(\phi)$ such that $Lit(n)$ is not set to false by σ. By definition C is acceptable. We need to show that C is a rl-cut in $G_h(\phi)$. Consider any rl-path π in $G_h(\phi)$. As there is no conflict in $G_h(\phi)$ due to σ at least one node m on π must not be set to false. By definition of C, $m \in C$. So removal of nodes in C from $G_h(\phi)$ disconnects the rl-path π. Since π is arbitrary, removal of nodes in C disconnects all rl-paths in the $G_h(\phi)$. This shows that C is a rl-cut.

(b) $\Rightarrow$ (a): Consider any rl-path π in $G_h(\phi)$. We will show that at least one node on π is not false. Since C is a rl-cut there must be at least one node $m \in C$ whose removal disconnects π. That is, $m \in \pi$. As C is acceptable $Lit(m)$ is not false. As π is arbitrary there is at least one node on each rl-path whose literal is not false. Thus, there can be no conflict. $\Box$

The following theorem formalizes the **case A** in Section 4.4.3. Intuitively, if there are two node-disjoint acceptable cuts in a hpgraph, then there cannot be any p-assigned node in the hpgraph. This in turn means that there cannot be any conflict or h-implied literals in the hpgraph (Corollaries 10, 11).

Theorem 5 *Given hpgraph $G_h(\phi)$, an assignment σ. The following are equivalent:*
(a) Let C_1, C_2 be two rl-cuts in $G_h(\phi)$ that are acceptable and node-disjoint ($C_1 \cap C_2 = \emptyset$).
(b) There is no node in $G_h(\phi)$ that is p-assigned due to σ.

Proof. (a) $\Rightarrow$ (b): Consider any rl-path π in $G_h(\phi)$. We will show that at least two nodes on π are not set to false by σ. Since C_1 is a rl-cut there must be a node n_1 that belongs to both C_1 and π. Similarly, there must be another node n_2 that belongs to both C_2 and π. Since C_1 and C_2 do not have any common nodes we know that $n_1 \neq n_2$. Also since both rl-cuts are acceptable $Lit(n_1), Lit(n_2)$ are not false. Thus, each rl-path in $G_h(\phi)$ has at least two distinct nodes whose literals are not set to false.

We now show (b) by contradiction. Assume there is a node n that is p-assigned due to σ. Then by definition there exists a rl-path $\pi \in G_h(\phi)$ such that all nodes m on π, $m \neq n$ have $Lit(m)$ as false. But this contradicts the fact that each rl-path has at least two distinct nodes whose literals are not set to false. Thus, there cannot be a p-assigned node n.

(b) $\Rightarrow$ (a): Observe that each rl-path π has at least two nodes that are not set to false by σ (otherwise, there will be a p-assigned node on π). Let $left(\pi)$ denote the leftmost node on π that is not set to false by σ and $right(\pi)$ denote the

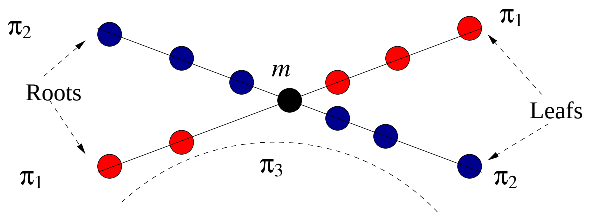


Figure 4.6: The rl-path π_3 is formed by combining rl-paths π_1 and π_2 at node m.

rightmost node on π that is not set to false by σ. Observe that both $left(\pi)$ and $right(\pi)$ exist and $left(\pi) \neq right(\pi)$.

We now define two set of nodes C_1 and C_2 as follows:

$$C_1 := \bigcup_{\pi \in G_h(\phi)} \{left(\pi)\}$$
$$C_2 := \bigcup_{\pi \in G_h(\phi)} \{right(\pi)\}$$

Observe that both C_1 and C_2 are rl-cuts and acceptable. We still need to show that C_1 and C_2 are node-disjoint. We use proof by contradiction. Assume there is a node m such that $m \in C_1$ and $m \in C_2$. Since $m \in C_1$ there exists an rl-path π_1 such that $left(\pi_1) = m$. Similarly, using the definition of C_2 it follows that there exists an rl-path π_2 such that $right(\pi_2) = m$.

Now consider a path π_3 obtained by concatenation of all nodes on π_1 until m, m, all the nodes on π_2 from m onwards (Figure 4.6). Observe that π_3 is an rl-path. Also observe that m is p-assigned due to π_3. This contradicts (b). Thus, it follows that C_1, C_2 are node-disjoint. $\square$

The following corollary formalizes **Case B** in Section 4.4.3. It states that if one of the watched rl-cuts, say C_1, is satisfied, then there cannot be any conflict or h-implied literal in the hpgraph. Thus, even if C_2 is no longer acceptable there is no need to update it.

Corollary 12 *Given hpgraph $G_h(\phi)$, an assignment σ. Let C be a rl-cut in $G_h(\phi)$ that is satisfied. Then the following holds:*
(a) There is no conflict in $G_h(\phi)$ due to σ.
(b) If n is a p-assigned node, then $n \in C$.
(c) There is no h-implied node in $G_h(\phi)$ due to σ.

Proof. (a) Observe that C is an acceptable rl-cut. Using theorem 4 the first claim follows easily.

(b) We will prove this claim using proof by contradiction. Suppose there exist a p-assigned node n and $n \notin C$. By definition, there exists an rl-path π such that for all $m \in \pi$ and $n \neq m$, $Lit(m)$ is false. Since C is a rl-cut there exists a common node n' such that $n' \in \pi$ and $n' \in C$. We assumed that $n \notin C$ so $n \neq n'$. Literal corresponding to every node on π (except n) is false, it follows that $Lit(n')$ must be false. This contradicts the fact that C is satisfied. Thus, $n \in C$.

(c) We use proof by contradiction. Suppose n is an h-implied node. By definition every h-implied node is also a p-assigned node. From (b) it follows that every p-assigned node is present in C. As C is satisfied $Lit(m)$ must be true for any

$m \in C$. Thus, $Lit(n)$ is true. This is a contradiction as $Lit(n)$ must be un-assigned according to the definition of an h-implied node. □

We partially formalize the **case D** in Section 4.4.3. This case arises when there is an acceptable cut in the hpgraph say C_1, but there is no other acceptable cut that is node-disjoint from C_1. The corollary below states that every p-assigned node must be contained in C_1. Thus, any h-implied literal will also be present in C_1.

Corollary 13 *Given hpgraph $G_h(\phi)$, an assignment σ. Let C_1 be a rl-cut in $G_h(\phi)$ that is acceptable. Suppose there is no other rl-cut in $G_h(\phi)$ that is both acceptable and node-disjoint from C_1. The following results can be derived from theorems 4,5.*

(a) There is no conflict in $G_h(\phi)$ due to σ.

(b) There is at least one p-assigned node in $G_h(\phi)$.

(c) Each p-assigned node n belongs to C_1 ($n \in C_1$).

(d) For each p-assigned node n, either $Lit(n)$ is already set to true or $Lit(n)$ is unassigned under σ . If $Lit(n)$ is unassigned it is a implied literal.

Proof. (a) C_1 is an acceptable rl-cut in $G_h(\phi)$. From theorem 4 it follows that there is no conflict in $G_h(\phi)$ due to σ.

(b) There are no two rl-cuts in $G_h(\phi)$ that are acceptable and node-disjoint. Thus from theorem 4 it follows that there is at least one p-assigned node in $G_h(\phi)$.

(c) Let n be a p-assigned node. By definition there exists an rl-path π such that for every node on $m \in \pi, m \neq n$, $Lit(m)$ is false. We will use proof by contradiction.

Suppose that $n \notin C_1$. Then there must be another node $n' \neq n$ such that $n' \in C_1$ and $n' \in \pi$ (as C_1 is a rl-cut). We know that literal corresponding to every node on π that is different from n is false. So $Lit(n') = false$. But this contradicts the fact that C_1 is acceptable (as $n' \in C_1$). Thus, $n \in C_1$.

(d) Let n be a p-assigned node. We know from (c) that $n \in C_1$. Since C is acceptable it follows that $Lit(n)$ cannot be false. $\square$

4.6 Minimal rl-cuts in Hpgraph

In the previous sections we generalized the CNF two-watched literal scheme to hpgraph by using two rl-cuts in the hpgraph $G_h(\phi)$. We now describe how rl-cuts are obtained and updated efficiently during BCP. The key idea is to make use of the vpgraph $G_v(\phi)$. For example, consider the hpgraph in Figure 4.7 (a) and the corresponding vpgraph in Figure 4.7 (b). Observe that any rl-path in the vpgraph corresponds to a rl-cut in the hpgraph. The rl-paths $\langle 1,3,4 \rangle$, $\langle 2,3,4 \rangle$, $\langle 5,6,8 \rangle$, $\langle 5,7,8 \rangle$ in the vpgraph corresponds to rl-cuts $\{1,3,4\}$, $\{2,3,4\}$, $\{5,6,8\}$, $\{5,7,8\}$, respectively, in the hpgraph.

Definition 14 *Given $G = (V,E,R,L,Lit)$ we say that $C \subseteq V$ is a* **minimal rl-cut** *in G iff C is a rl-cut in G and no proper subset of C is a rl-cut in G. Let nodes(π) denote the set of nodes occurring on a rl-path π.*

A surprising fact is that the rl-paths in the vpgraph correspond to minimal rl-cuts in the hpgraph. One can also prove that every minimal rl-cut in the hp-

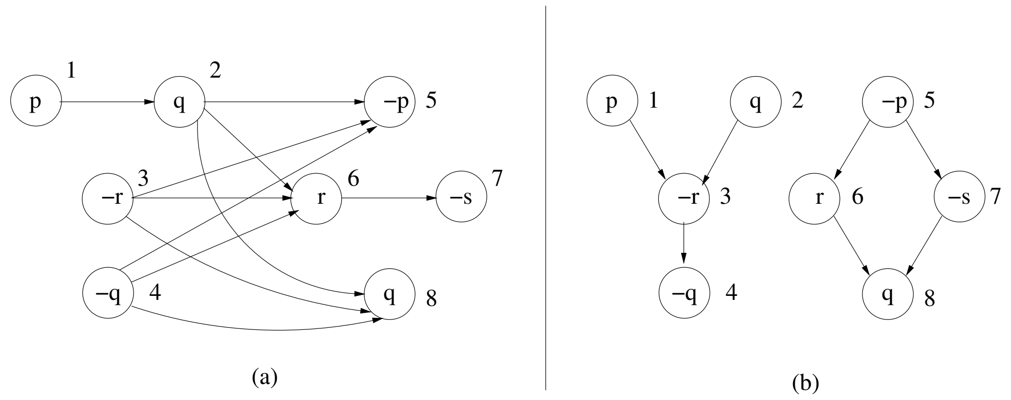


Figure 4.7: (a) Hpgraph for formula $(((p \vee q) \wedge \neg r \wedge \neg q) \vee (\neg p \wedge (r \vee \neg s) \wedge q))$. (b) The corresponding vpgraph.

graph corresponds to a rl-path in the vpgraph. This *duality* between rl-cuts in the hpgraph and rl-paths in the vpgraph is formalized below.

Theorem 6 *Given hpgraph $G_h(\phi)$ and vpgraph $G_v(\phi)$ for a formula ϕ. Let π be a rl-path in $G_v(\phi)$. Then nodes(π) form a minimal rl-cut in $G_h(\phi)$.*

Theorem 7 *Given hpgraph $G_h(\phi)$ and vpgraph $G_v(\phi)$ for a formula ϕ. Let C be a minimal rl-cut in $G_h(\phi)$. Then there exists a rl-path π in $G_v(\phi)$ such that $C = nodes(\pi)$.*

We give the proofs for theorems 6,7 in the appendix C. One can prove similar duality between rl-paths in the hpgraph and rl-cuts in the vpgraph.

Theorem 8 *Given vpgraph $G_v(\phi)$ and hpgraph $G_h(\phi)$ for a formula ϕ. Let π be a rl-path in $G_h(\phi)$. Then nodes(π) form a minimal rl-cut in $G_v(\phi)$.*

Theorem 9 *Given vpgraph $G_v(\phi)$ and hpgraph $G_h(\phi)$ for a formula ϕ. Let C be a minimal rl-cut in $G_v(\phi)$. Then there exists a rl-path π in $G_h(\phi)$ such that $C = nodes(\pi)$.*

4.6.1 Finding and Updating Minimal rl-cuts in Hpgraph

Our algorithm always maintains two minimal rl-cuts in the hpgraph as the watched cuts. These cuts are obtained and updated by finding rl-paths in the corresponding vpgraph by using a depth-first search like routine. The BCP algorithm relies on the ability to find acceptable rl-cuts in the hpgraph. This is done by searching for acceptable rl-paths in the vpgraph[3]. We always search the vpgraph for acceptable rl-paths. We omit the qualifier "acceptable" in the following.

The BCP routine also requires that we find disjoint rl-cuts in the hpgraph (if possible). This is done by searching for disjoint rl-paths in the vpgraph. More precisely, suppose we are trying to replace rl-cut C_2 in the hpgraph. Let the other rl-cut in the hpgraph be C_1 and let π_1 denote the rl-path corresponding to C_1 in the vpgraph. Then we search for a rl-path in the vpgraph that is completely disjoint from π_1. If we succeed in finding a path π_2 in the vpgraph that is completely disjoint from π_1, then we obtain a replacement C_3 for C_2 in the hpgraph such that $C_1 \cap C_3 = \emptyset$. If there is no rl-path in the vpgraph that is completely disjoint from π_1, we find the set of all nodes N on π_1 that must be shared by any acceptable rl-path in the vpgraph. Intuitively, the nodes in N are precisely the p-assigned nodes and give us the exact set of h-implied literals.

[3]A rl-path π is acceptable if no node on π is falsified under the current assignment.

Theorem 10 *Given vpgraph $G_v(\phi)$ and hpgraph $G_h(\phi)$ for a formula ϕ. Let π_1 be an acceptable rl-path in $G_v(\phi)$. Suppose there is no other acceptable rl-path in $G_v(\phi)$ that is completely node disjoint from π_1. Let N denote the set of nodes that must be shared in any acceptable rl-path in $G_v(\phi)$. We have the following results:*
(a) Every p-assigned node in $G_h(\phi)$ belongs to N.
(b) $N \neq \emptyset$.
(c) Every node in N is p-assigned in $G_h(\phi)$.

Proof. (a) Every acceptable rl-path π_i in $G_v(\phi)$ corresponds to a minimal rl-cut C_i in $G_h(\phi)$. Since we do not have acceptable and node-disjoint rl-paths in $G_v(\phi)$, we cannot have acceptable and node-disjoint rl-cuts in $G_h(\phi)$. From corollary 13 it follows that each p-assigned node belongs to each C_i. Thus, each p-assigned node belongs to $\cap_i C_i = \cap_i \pi_i = N$.

(b) From (a) we know that every p-assigned node belongs to N. From corollary 13 there exists at least one p-assigned node in $G_h(\phi)$. Thus, $N \neq \emptyset$.

(c) Consider a node $n \in N$. We want to show that n is p-assigned in $G_h(\phi)$. Let M denote the set of all nodes in $G_v(\phi)$ which are false under the current assignment. Let $M' = M \cup \{n\}$. We claim that M' is a rl-cut in $G_v(\phi)$. In order to show this we will show that every rl-path in $G_v(\phi)$ gets disconnected if the nodes in M' are removed from $G_v(\phi)$. There are two possibilities for a rl-path π in $G_v(\phi)$. 1) π is an acceptable rl-path in $G_v(\phi)$. By definition of N we have $n \in \pi$. So removal of $n \in M'$ will disconnect π. 2) π is not an acceptable rl-path in $G_v(\phi)$. Then there must exist a node $m \in \pi$ such that $Lit(m)$ is false. By definition of M we have $m \in M$ so $m \in M'$. Thus, M' is a rl-cut in $G_v(\phi)$.

Observe that M' is not necessarily a minimal rl-cut. Let $M'' \subseteq M$ represent a minimal rl-cut obtained by removing nodes from M' (M'' is not necessarily unique). We claim that $n \in M''$. This is because n is the only node in M' that disconnects all acceptable rl-paths in $G_v(\phi)$. Thus, n must be present in any minimal rl-cut obtained from M'.

Now we use the duality between the minimal rl-cuts in $G_v(\phi)$ and rl-paths in $G_h(\phi)$. The minimal rl-cut M'' in $G_v(\phi)$ corresponds to an rl-path π_h in $G_h(\phi)$ such that $M'' = nodes(\pi_h)$. By definition of M'' it follows that for every $m \in \pi_h$, $m \neq n$, $Lit(m)$ is false. Thus, node n is p-assigned due to π_h. $\square$

4.6.2 Implementation of the Cut Replacement Algorithm

Our solver always maintains minimal rl-cuts in the hpgraph and we omit the qualifier "minimal" in the following. Suppose we are trying to replace rl-cut C_2 in the hpgraph. Let the other rl-cut in the hpgraph be C_1 and let π_1 denote the rl-path corresponding to C_1 in the vpgraph.

The first step is to search for an acceptable rl-path in the vpgraph that is completely disjoint from π_1. This is done by performing depth first search (DFS) in the vpgraph and ignoring any nodes that are falsified or lie on π_1. If the DFS routine encounters a leaf node, then we can immediately produce an acceptable rl-path π_2 in the vpgraph that is completely disjoint from π_1. This can be done by following the parent nodes starting from the leaf node.

If the DFS algorithm fails to reach a leaf node, then it means that there is no acceptable rl-path in the vpgraph that is completely disjoint from π_1. In this case we try to find an acceptable rl-path in the vpgraph that is as disjoint from π_1 as possible. In other words, we try to find an acceptable rl-path π_2 such that the intersection of π_1 and π_2 is exactly the set of p-assigned nodes (the set N in theorem 10). This is the second step of the cut replacement algorithm. Our implementation of the second step is described below.

- We identify a subgraph $G'_v(\phi)$ of vpgraph $G_v(\phi)$ such that each rl-path in $G'_v(\phi)$ is acceptable. This can be done by performing DFS on $G_v(\phi)$ and removing nodes with false literals and removing non-leaf nodes with no children. In the actual implementation we do not modify $G_v(\phi)$, instead we keep a flag with each node indicating whether the node is in $G'_v(\phi)$. If $G'_v(\phi)$ is empty then it means that there is no acceptable rl-path in $G_v(\phi)$. In this case we report a conflict. If $G'_v(\phi)$ is not empty, then we perform the following steps.

- We identify a subset of nodes in $G'_v(\phi)$ whose removal disconnects all rl-paths in $G'_v(\phi)$. These nodes are exactly the intersection of all rl-paths in $G'_v(\phi)$ or the set N in theorem 10. This is done by a modification of breadth first search on $G'_v(\phi)$. We maintain a frontier of nodes in $G'_v(\phi)$. Initially, the frontier contains the roots in $G'_v(\phi)$. In each iteration we remove a node from the frontier that occurs earliest in the topologically sorted order of nodes in $G'_v(\phi)$ and insert its children in the frontier. If at any iteration the frontier contains a single node n, it means that all rl-paths in $G'_v(\phi)$ must go through n. Thus, n is a p-asssigned node and it is added to the set of p-assigned nodes. The set of p-assigned nodes at the end of the algorithm is the set N in theorem 10.

- We find an acceptable rl-path in the vpgraph that is as disjoint from π_1 as possible. This is done by perfoming DFS on $G'_v(\phi)$ and ignoring any node that occurs on π_1 but not in N (we need to take nodes from N). The result is a rl-path π_2 in $G_v(\phi)$ such that $\pi_1 \cap \pi_2 = N$.

4.7 Two-watched rl-cuts for each Hpgraph Component

In the previous sections we described how the two-watched rl-cuts in the hpgraph can be used to carry out Boolean constraint propagation. BCP based on *only* two-watched rl-cuts can be very inefficient when the hpgraph has millions of nodes. This is because even the minimal rl-cuts for the entire hpgraph can be large and will be updated frequently during BCP (see Figure 4.8(a)).

If the input NNF formula ϕ is a conjunction of a number of smaller sub-formulas $\phi_1, \ldots, \phi_k$, then $G_h(\phi)$ is a disjoint union of $G_h(\phi_1), \ldots, G_h(\phi_k)$. We refer to $G_h(\phi_i)$ as an *hpgraph component*. In practice, there are usually many hpgraph components and each hpgraph component is small as compared to the entire hpgraph in terms of number of nodes [4]. In our implementation we maintain

[4]We can also control the number and the size of hpgraph components by introducing new variables in ϕ.

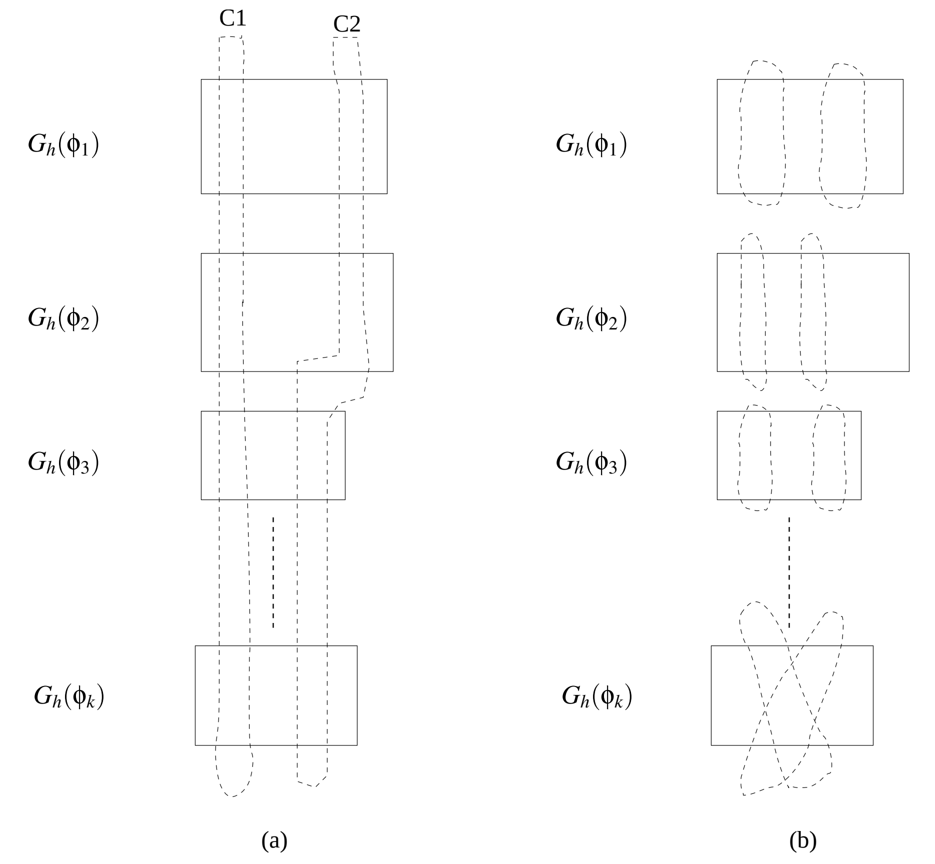


Figure 4.8: (a) Two monolithic rl-cuts for the entire hpgraph. (b) Two rl-cuts for each hpgraph component.

two-watched rl-cuts for each hpgraph component (see Figure 4.8(b). This allows more locality during BCP as the rl-cuts for an hpgraph component can be updated locally by looking only at the nodes in the hpgraph component. Note that the BCP algorithm and the results discussed earlier for the hpgraph apply to each individual hpgraph component and the corresponding vpgraph component.

4.8 Relationship with the CNF Two-watched Literal Scheme

The hpgraph for a CNF formula is a disjoint union of various line graphs (hpgraph components) where each line graph represents a clause. A minimal rl-cut in a line graph is simply a cut of size one. Thus, the two-watched rl-cuts for each hpgraph component reduces to two-watched literal scheme when the input is a CNF formula (see Figure 4.9). We can show that various steps in our BCP algorithm for updating watched rl-cuts reduce to updating the watched literals when the hpgraph represents a CNF formula. Thus, we consider the two-watched rl-cuts scheme for hpgraph as a generalization of the CNF two-watched literal scheme.

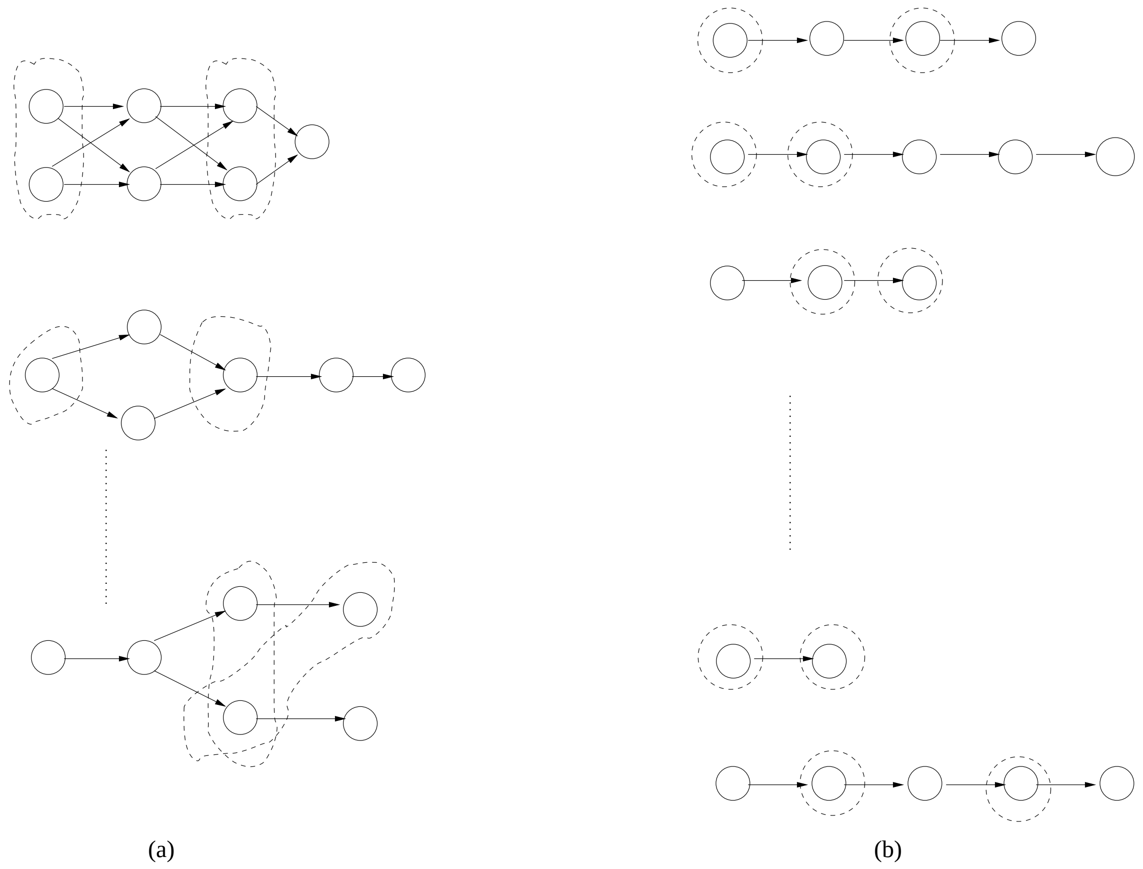


Figure 4.9: (a) General form of a hpgraph with two-watched rl-cuts for each hpgraph component. (b) The hpgraph for a CNF formula. The two-watched cut scheme reduces to two-watched literal scheme.

4.9 Other Aspects of our SAT Solver

The other important components of our DPLL based SAT solver such as decision heuristics, conflict driven learning, non-chronological backtracking, and restarts are implemented in a similar manner as other state-of-the-art DPLL based SAT solvers.

Various optimizations are used for improving the performance of our solver. These optimizations are described below.

- The conflict driven learning generates new clauses, which are added to a CNF clause database. The BCP routine takes into account both the hpgraph and the clause database in order to detect conflicts and implied literals.

- When converting a Boolean circuit to NNF we introduce new variables for gates with fanout greater than one. This usually produces a large number of small hpgraph components (less than hundred nodes) and a few very large hpgraph components. We try to avoid large hpgraph components by adding extra new variables when converting a Boolean circuit to a NNF formula. The intuition is that we can introduce a new variable to cut a particular (large) sub-tree from a Boolean circuit.

- The hpgraph components containing a few clauses are removed and the clauses contained in such components are added to the clause database before the main DPLL algorithm begins.

- Let n be an h-implied node ($Lit(n)$ is an implied literal). We do not explicitly store the clause due to which $Lit(n)$ is implied. Instead we simply record that $Lit(n)$ was implied at node n in the hpgraph. The actual reason due to which $Lit(n)$ is implied is useful only during conflict analysis. This reason is computed on demand during the conflict analysis.

4.10 Experimental Results

The experiments are performed on a 1.86 GHz Intel Xeon (R) machine with 4 GB of memory running Linux. The techniques described in the chapter have been implemented in a SAT solver called NFLSAT (Non-clausal FormuLas SATisfiability checker). The input formula is given in AIG (And Inverter Graph) [1], or ISCAS format.

4.10.1 Benchmarks

We evaluate the solver on a collection of 2541 Boolean circuits obtained from publicly available sources. We describe the sources of these benchmarks below.

- K-induction benchmarks (AIG format): We generated 857 k-induction [130] problems from sequential circuits used in the 2007 hardware model checking competition [7]. We used the publicly available utilities `aigtosmv` [1] and `smv2qbf` [19] in order to generate the k-induction benchmarks. `smv2qbf` was run with the options `-i -aig`.
- Bounded model checking (BMC) benchmarks (AIG format): We generated

839 bounded model checking problems using the sequential circuits from [7]. We used the publicly available utility `aigbmc` [1] in order to generate BMC problems.

- SAT competition 2007 benchmarks (AIG format): We use all 341 benchmarks that were used in the AIG track in SAT competition 2007 [15]. Around 220 of these benchmarks were generated by extracting the circuit structure from CNF instances using `cnf2aig`. The CNF instances themselves were obtained from multiple domains such as software and hardware verification, cryptography, and planning. The remaining benchmarks consist of 105 k-induction benchmarks and 16 benchmarks generated using `c32sat` [47].
- UCLID benchmarks (ISCAS format): We used 56 benchmarks generated by the UCLID tool [22]. These were provided to us by Sanjit Seshia.
- Equivalence checking benchmarks (ISCAS format): We use 71 benchmarks from equivalence checking domain.
- Microprocessor verification benchmarks (ISCAS format): We use 222 benchmarks (fvp-unsat.2.0-iscas, sss-sat-1.0-iscas, vliw-sat-1.1-iscas) made available by M.N. Velev [10].
- Other benchmarks: Around 48 benchmarks were obtained by extracting circuit structure from CNF instances using `cnf2aig`. Another 105 benchmarks are k-induction benchmarks generated using different levels of AIG optimizations [127, 1].

Solver	Solved	Failed	Solved Time	Total Time
NFLSAT	**2364**	**177**	**29753**	**135953**
RSAT	2310	231	45794	184394
PicoSAT	2281	260	43297	199297
MiniSAT	2270	271	39489	202089

Table 4.1: Comparison between SAT solvers.

4.10.2 Comparison with SAT 2007 Competition Winners

We compare NFLSAT against three state-of-the-art CNF solvers RSAT [14], PicoSAT [13], and MiniSat [8]. In SAT 2007 competition the solvers RSAT, PicoSAT, and MiniSAT were ranked first, second, third, respectively in the industrial category. We use the SAT 2007 competition version of RSAT and PicoSAT. We use an updated version of MiniSAT (minisat2-070721) available from [8]. (This version of MiniSAT is approximately 20% faster than the SAT 2007 competition version of MiniSAT on our benchmarks.)

The (equi-satisfiable) CNF versions of the above circuits were obtained by means of basic Tseitin transformation [138, 124]. We use `aigtocnf` to convert the benchmarks in AIG format to CNF. The benchmarks in ISCAS format were converted to CNF by introducing a new variable for each gate in the circuit. We do not include the time required to convert a Boolean circuit to CNF in the run times reported below.

The experimental results are summarized in Table 4.1. There was a timeout of 10 minutes per problem per solver. For each solver we report the following quantities: 1) The total number of problems out of 2541 problems that were solved within timeout in the ”Solved” column. 2) The total number of problems that the solver could not solve due to a timeout or a memory out in the ”Failed” column. 3) The total time spent in seconds on the problems that were solved in the ”Solved Time” column. 4) The sum of ”Solved Time” and the time spent on failed problems in the ”Total Time” column.

NFLSAT solves more problems than each of the CNF SAT solvers and it is also faster in terms of run time. Intuitively, the better performance of NFLSAT is because of the following: 1) The NNF form of Boolean circuits has fewer variables than (pre-processed) CNF in the majority of the cases (see Chapter 2). Fewer variables in turn reduce the overhead during the BCP and can make the decision heuristics more effective. 2) The two watched-cut scheme carries more overhead than the two-watched literal scheme. However, the two-watched cut scheme can potentially update the watches for a large number of clauses without having to look at each clause individually. 3) Optimizations for efficient BCP on the clause database [4, 9].

Due to implementation differences between various solvers it is extremely hard to pin point the reason for the better performance of NFLSAT. Even minor implementation differences can make the solvers explore different search spaces leading to significant differences in run time. Figures 4.10, 4.11, 4.12 give scatter plots comparing NFLSAT with RSAT, PicoSAT, MiniSAT, respectively. For each CNF solver we compare it with NFLSAT on all instances, satisfiable instances, and unsatisfiable instances.

The main conclusion is that NFLSAT is competitive to existing state-of-the-art

Solver	Solved	Failed	Solved Time	Total Time
NFLSAT	**2060**	**132**	**26585**	**105785**
MiniSAT++	2074	118	32457	103257
PicoaigerSAT	2033	159	35892	131292

Table 4.2: Comparison of NFLSAT with SAT-Race 2008 AIG track winners.

CNF SAT solvers. The two-watched cut scheme and the use of vpgraph enables efficient BCP on hpgraph. While we have have carefully implemented and optimized NFLSAT, the implementation is still not as mature as CNF solvers which have been optimized over the past seven years. There is still scope for implementation and heuristics improvement in our solver.

4.10.3 Comparison with SAT-Race 2008 AIG Track Winners

We compare NFLSAT with MiniSAT++ 1.0 which was ranked first in the AIG track of SAT-Race 2008. MiniSat++ simplifies the AIG circuit using DAG-aware rewriting and then converts the simplified circuit to CNF by using an improved Tseitin translation [9, 76]. The resulting CNF is then passed to MiniSAT 2.1, which was ranked first in the CNF track in SAT-Race 2008. We also compare NFLSAT with PicoaigerSAT which was ranked second in the AIG track of SAT-Race 2008. We evaluate the three solvers on a collection of 2192 AIG benchmarks[5].

The experimental results are summarized in Table 4.2. There was a timeout of

[5]Unlike SAT competitions the participants of SAT-Race are not required to make their solvers (source code or binary) publicly available. We obtained MiniSAT++ binary and PicoaigerSAT source code from their authors.

10 minutes per problem per solver. For each solver we report the following quantities: 1) The total number of problems out of 2192 problems that were solved within timeout in the ”Solved” column. 2) The total number of problems that the solver could not solve due to a timeout or a memory out in the ”Failed” column. 3) The total time spent in seconds on the problems that were solved in the ”Solved Time” column. 4) The sum of ”Solved Time” and the time spent on failed problems in the ”Total Time” column. Figures 4.13, 4.14 give scatter plots comparing NFLSAT with MiniSAT++, PicoaigerSAT respectively.

NFLSAT solves more problems than PicoaigerSAT and is also faster in terms of run time. MiniSAT++ solves 14 more problems than NFLSAT. However, on majority of the benchmarks NFLSAT is faster than MiniSAT++ as shown by the scatter plot in Figure 4.13.

Detailed comparison with MiniSAT++

We divide our collection of AIG benchmarks in three sets: 1) K-IND set consists of K-induction benchmarks. 2) BMC set consists of BMC benchmarks. 3) SAT-2007 benchmarks consists of AIG benchmarks used in AIG track of SAT 2007 competition. Around 220 SAT-2007 benchmarks were obtained by extracting circuit structure from CNF using `cnf2aig`.

The experimental results are shown in Table 4.3. The column ”#Probs” gives the number of problems in each benchmark set. There was a timeout of 10 minutes per problem per solver. For each solver we report the following quantities: 1) ”Solved” gives the total number of problems that were solved within timeout. 2)

Benchmarks	#Probs	NFLSAT			MiniSat++ 1.0		
		Solved	Solved time	Total time	Solved	Solved Time	Total time
K-IND	857	842	3719	12719	840	12496	22696
BMC	838	822	4786	14386	823	5623	14623
SAT-2007	343	245	14555	67955	259	12204	57204

Table 4.3: Comparison between NFLSAT and MiniSAT++ 1.0.

”Solved time” gives the total time spent on solved problems. 3) ”Total time” is the sum of ”Solved time” and the time spent on problems where a timeout occurred.

On K-IND benchmarks NFLSAT solves two more problems and is 3.36 times faster than MiniSAT++ in terms of time spent on solved problems. On BMC benchmarks MiniSAT++ solves one more problem than NFLSAT. The runtimes are similar. On SAT-2007 benchmarks MiniSAT++ solves 14 more problems than NFLSAT. The poor performance of NFLSAT on SAT-2007 benchmarks is on the circuits that were obtained from CNF formulas. The extraction of circuit structure from CNF is not perfect and many of the extracted circuits are simply a conjunction of clauses. Figures 4.15, 4.16, 4.17 give scatter plots comparing NFLSAT with MiniSAT++ on K-IND, BMC, SAT-2007 benchmark sets, respectively.

Note that NFLSAT currently does not employ the idea of DAG-aware minimization that MiniSAT++ employs. Adding this idea to NFLSAT is likely to improve the performance of NFLSAT.

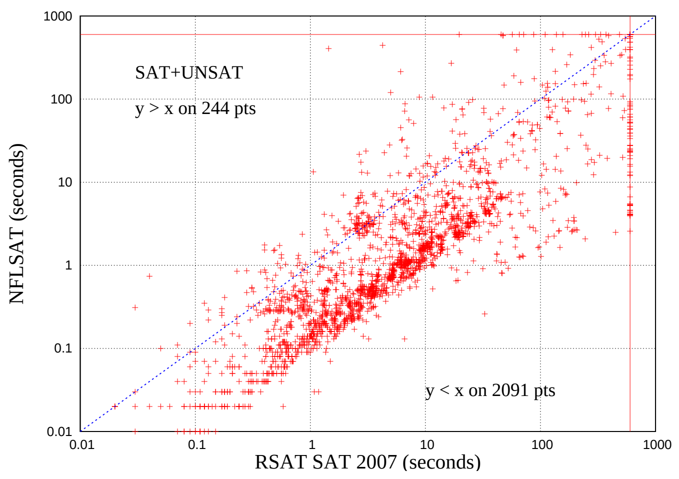
4.10.4 Breakdown of the Total Time

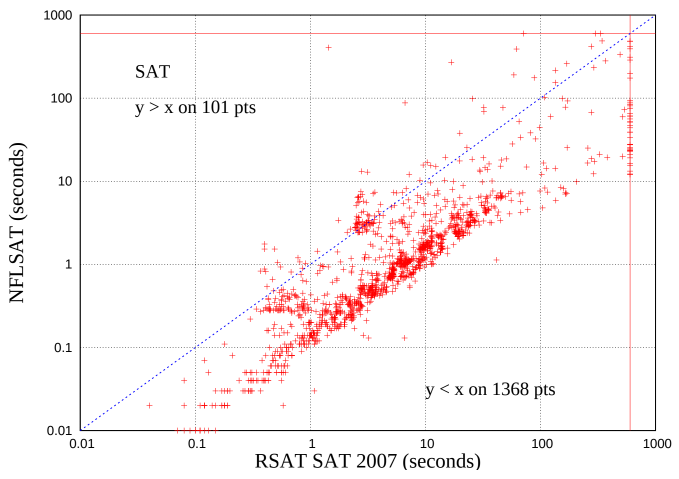
The frontend for NFLSAT performs the following tasks: 1) Read the input formula/circuit. 2) Obtain the NNF form from the given circuit by introducing new variables. 3) Obtain the hpgraph and vpgraph from the NNF form. 4) Set up all data structures that are to be used in the main DPLL algorithm. 5) Perform top level Boolean constraint propagation (without making any decisions). Figure 4.18 compares the time taken by the frontend (y-axis) with the total number of AND gates in the input AIG circuit (x-axis). It can be seen that the frontend scales polynomially with the input size. In particular, there is no blowup involved in constructing hpgraph and vpgraph.

The DPLL time denotes the time taken by the actual DPLL algorithm. It is obtained by subtracting the frontend time from the total time. Figure 4.19 compares the time taken by the frontend (x-axis) to the DPLL time (y-axis). The frontend time exceeds the time taken by DPLL algorithm on many benchmarks. However, the frontend time is less than ten seconds for the majority of cases.

4.11 Chapter Summary

We presented a DPLL based SAT solver that operates on the graph based representations of NNF formulas. The hpgraph encodes the CNF form of a given NNF formula, while the vpgraph encodes the DNF form of the given NNF formula. The key step in the DPLL algorithm is Boolean constraint propagation (BCP). We generalize the idea of two-watched literal scheme from CNF SAT solvers in order to efficiently carry out BCP on hpgraph. In our algorithm two cuts are watched for each hpgraph component. The watched cuts are used to detect conflicts and implied literals. We use the duality between the cuts in a hpgraph and the paths in a vpgraph for efficiently updating the cuts. Experimental results show that the new SAT solver is faster than the state-of-the-art solvers on majority of the benchmarks and is competitive in terms of the number of problems solved.





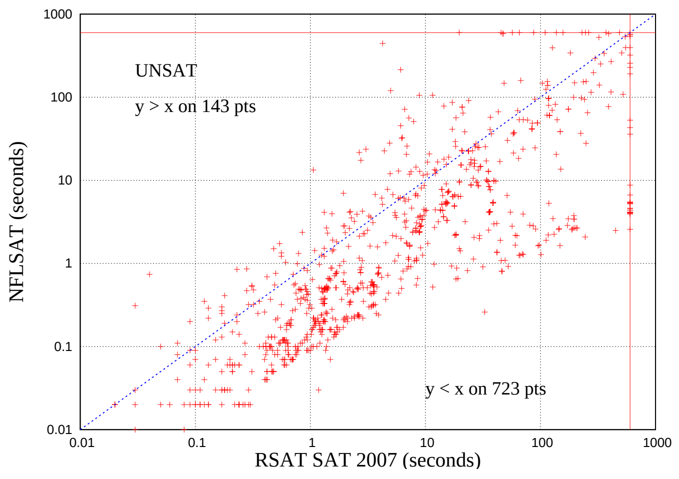


Figure 4.10: Scatter plot comparing the run times of NFLSAT and RSAT.

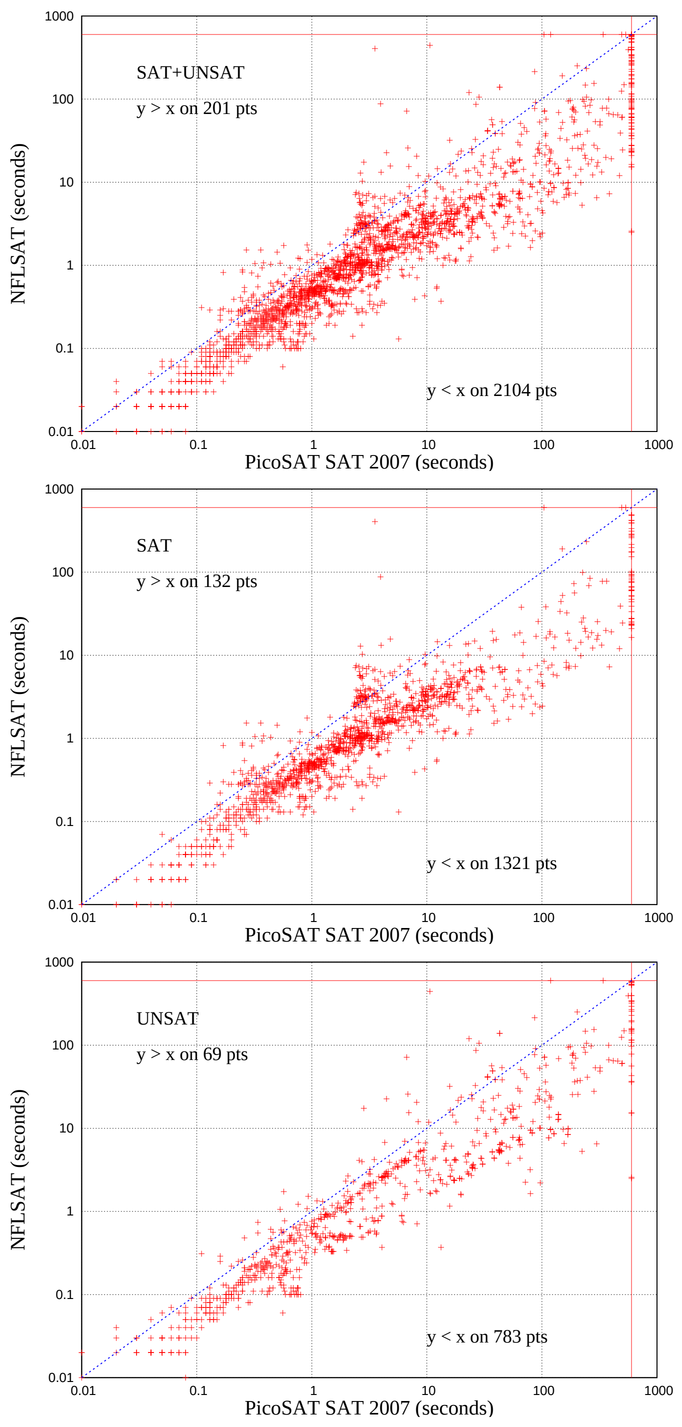


Figure 4.11: Scatter plot comparing the run times of NFLSAT and PicoSAT.

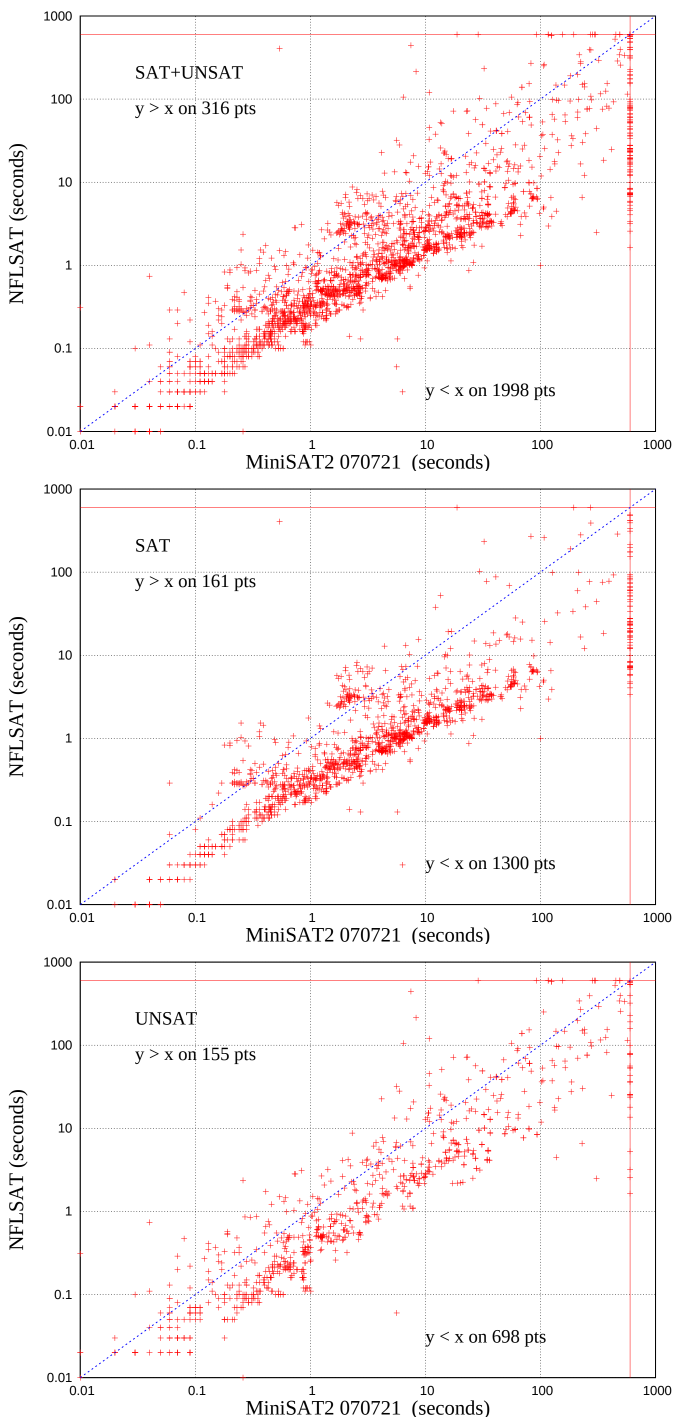
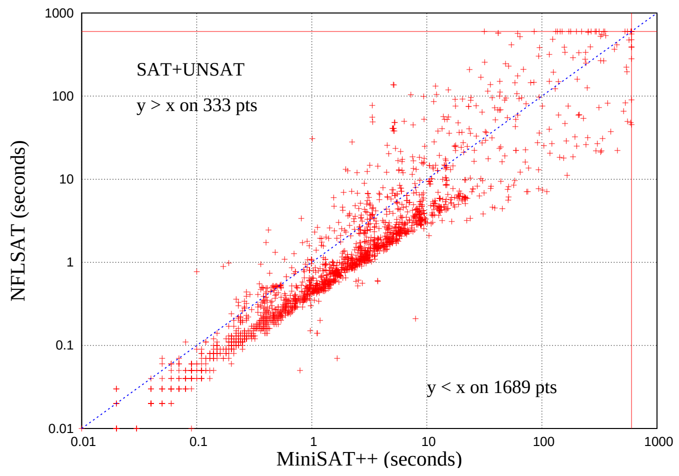
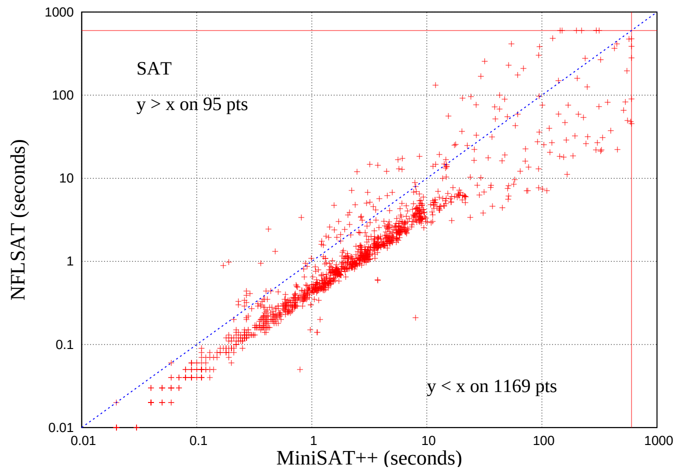


Figure 4.12: Scatter plot comparing the run times of NFLSAT and MiniSAT.





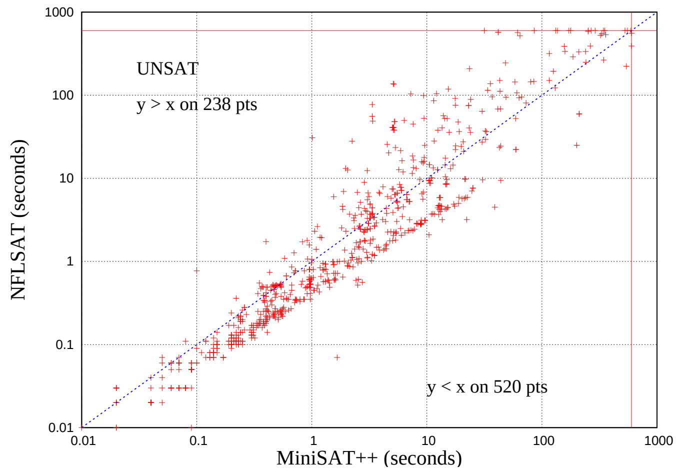
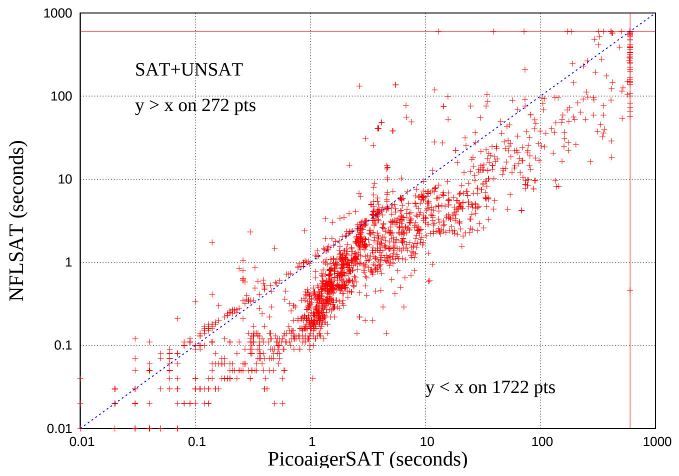
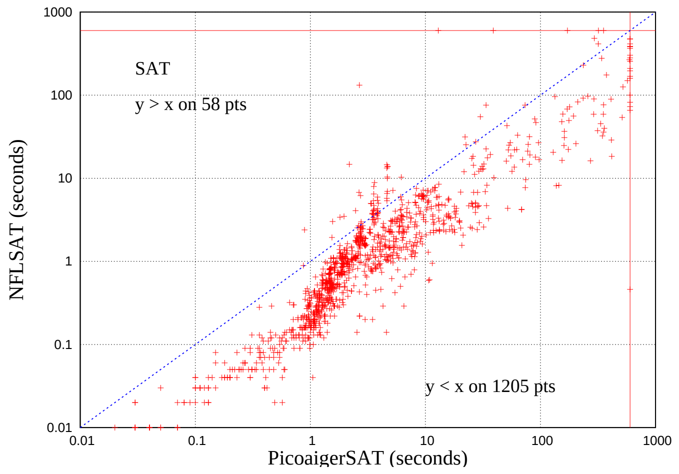


Figure 4.13: Scatter plot comparing the run times of NFLSAT and MiniSAT++.





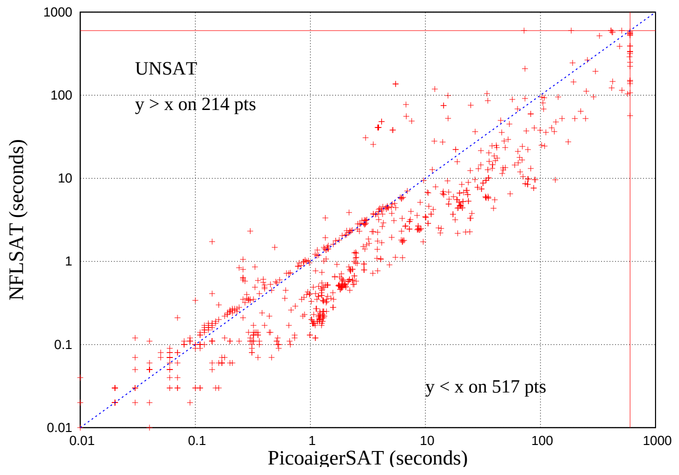


Figure 4.14: Scatter plot comparing the run times of NFLSAT and PicoaigerSAT.

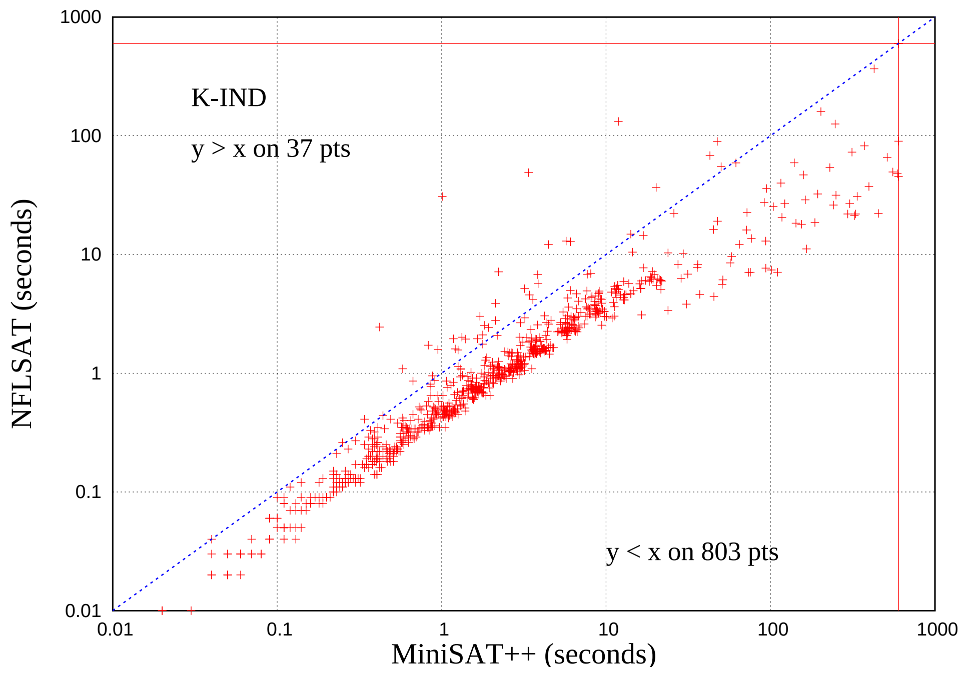


Figure 4.15: Scatter plot comparing the run times of NFLSAT and MiniSAT++ on K-induction benchmarks.

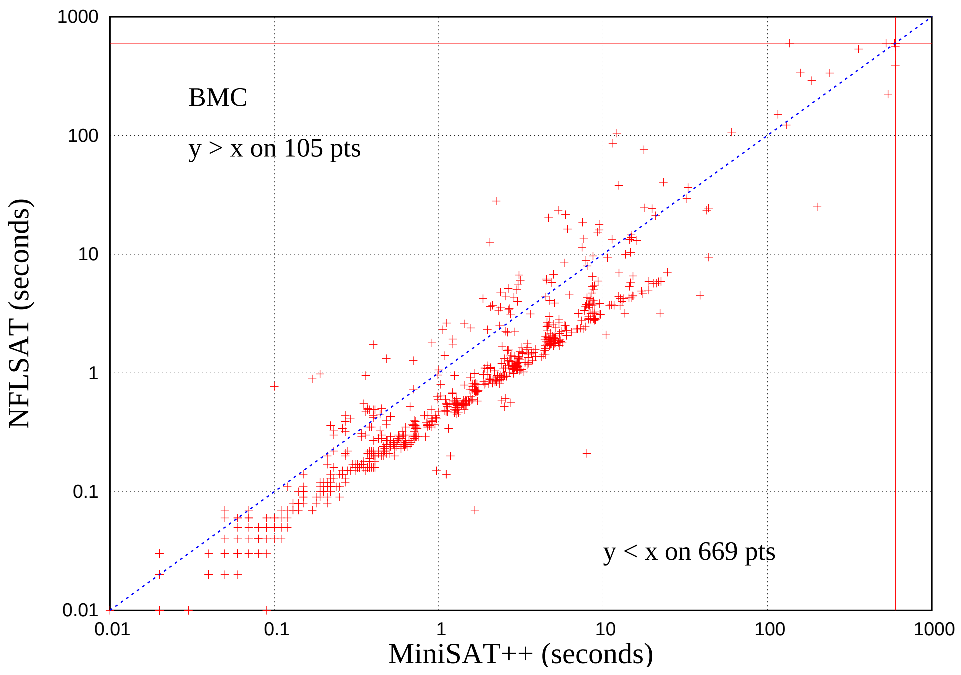


Figure 4.16: Scatter plot comparing the run times of NFLSAT and MiniSAT++ on BMC benchmarks.

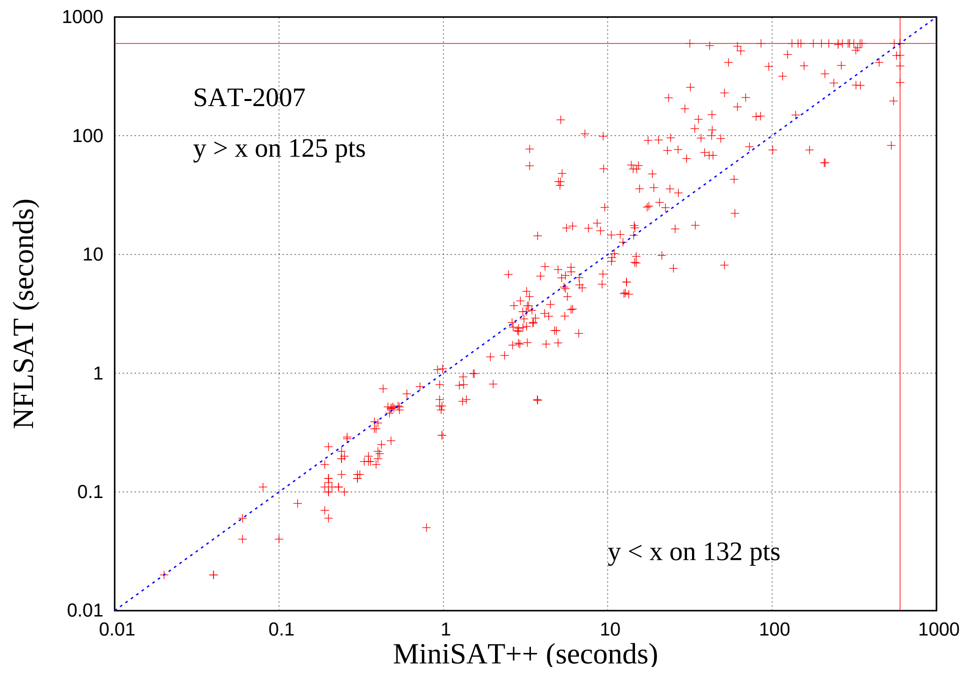


Figure 4.17: Scatter plot comparing the run times of NFLSAT and MiniSAT++ on SAT-2007 AIG benchmarks.

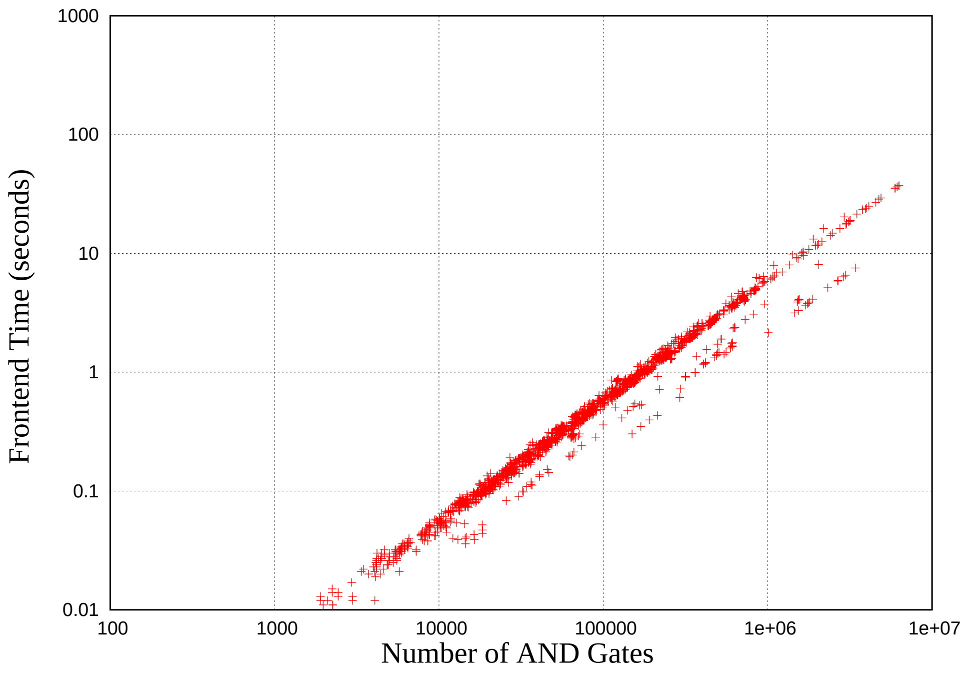


Figure 4.18: The frontend time as a function of circuit size (measured in number of AND gates).

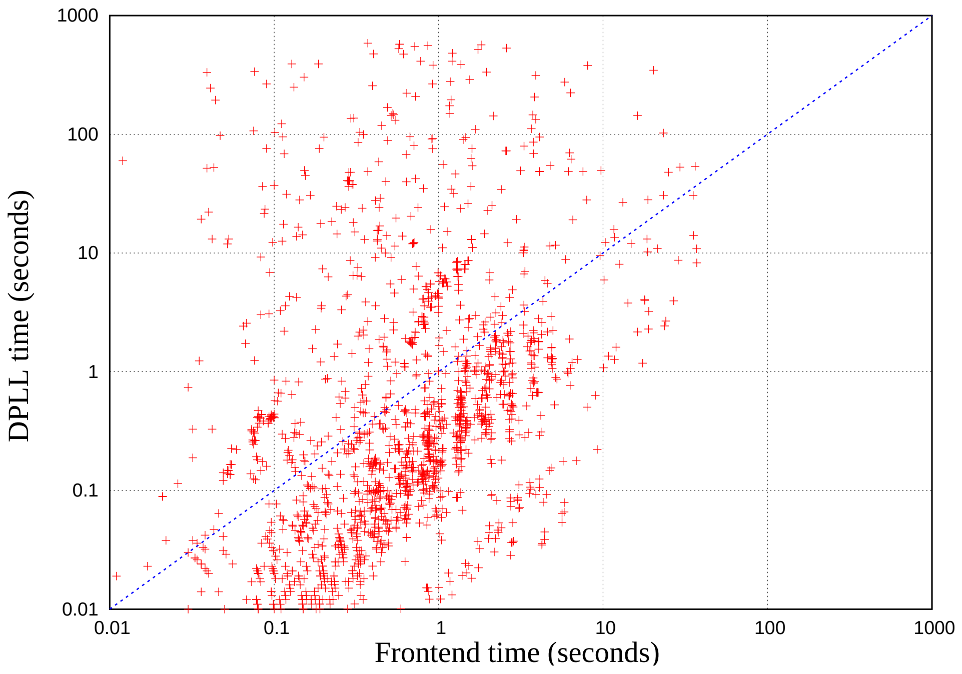


Figure 4.19: Frontend time and the time spent in the DPLL algorithm.

Chapter 5

Techniques for Word-Level Verification

In the software domain, one of the successful abstraction technique for large systems is *predicate abstraction* [85]. It abstracts data by only keeping track of certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. Predicate abstraction of ANSI-C programs in combination with counterexample guided abstraction refinement was introduced by Ball and Rajamani [34, 33] and promoted by the success of the SLAM project. The goal of this project is to verify that Windows device drivers obey API conventions. The abstraction of the program [37, 35] is computed using a theorem prover such as Simplify [73] or Zapato [36].

In this work we use predicate abstraction for verifying hardware designs. Predicate abstraction is only effective if the predicates can cover the relationship between registers (multiple latches). This typically requires a word-level model given at the RT-level of a hardware description language. RT-level models are similar to programs written in a programming language, such as ANSI-C. We apply predicate abstraction to word-level models given in RTL Verilog.

Many software verification tools use theorem provers for computing the predicate abstraction. Theorem provers typically model the variables using unbounded integer numbers. Overflow and bit-wise operators are not modeled. However, hardware description languages like Verilog provide an extensive set of bit-wise operators. For hardware designs, the use of these bit-level constructs is ubiquitous. As in [103, 59, 90, 62], we use a bit-level SAT solver to compute the abstract transition relation. This allows us to precisely model the bit-vector semantics of hardware designs during abstraction computation.

We view our technique as a word-level verification technique because of the following: 1) the predicates that are used for computing the predicate abstraction are at the word-level [1], and 2) the use of a bit-level SAT solver as a decision procedure can be replaced by a word-level solver. However, existing word-level solvers for hardware description languages are not always competitive with bit-level SAT solvers.

[1] If needed bit-level predicates can be used as well. For example, a predicate of the form `rg[2]` is allowed where `rg` is a register and `rg[2]` refers to the second bit in `rg`.

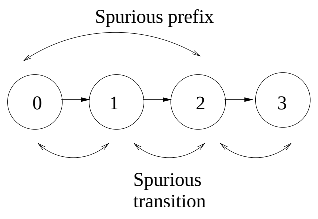


Figure 5.1: A spurious counterexample.

5.1 Contributions

This thesis applies predicate abstraction and refinement for verifying circuits given in Verilog RTL. Two problems arise when applying predicate abstraction to circuits: 1) The computation of the abstract model in presence of a large number of predicates, and 2) discovery of suitable word-level predicates for abstraction refinement.

In order to address the first problem, we divide the set of predicates into *clusters* of related predicates. The abstraction is computed separately with respect to the predicates in each cluster. Since each cluster contains only a small number of predicates, the computation of the abstraction becomes more efficient. We refer to this technique as *predicate clustering*. It allows us to tune the abstraction step between the two extremes of eager abstraction [59] and lazy abstraction [88] . The eager technique refers to the case where all predicates are within a single cluster, while lazy abstraction corresponds to the case in which many clusters of small cardinality (size) are used for computing the abstraction.

When refining the abstract model using a spurious counterexample, we distinguish between two cases of spurious behavior [63]: *Spurious transitions* are abstract transitions that do not have any corresponding concrete transitions. By definition, spurious transitions cannot appear in the most precise predicate abstraction, which is computed by the eager approach. However, predicate clustering usually produces coarse abstractions, which can give rise to spurious transitions. *Spurious prefixes* are prefixes of the spurious counterexample that do not have a corresponding concrete path. This happens when the set of predicates is not rich enough to capture the relevant behaviors of the concrete system, even for the most precise abstraction. Fig. 5.1 shows a spurious counterexample containing a spurious transition and a spurious prefix.

When a spurious counterexample is encountered, we first check whether each transition in the counterexample can be simulated on the original program. This is done by creating a SAT instance for the simulation of each abstract transition. If the SAT instance for an abstract transition is unsatisfiable, then the abstract transition is spurious. In this case, we refine the abstraction by adding constraints on the abstract transition relation that eliminate the spurious transition. We make use of the proof of unsatisfiability of the SAT instance to identify a small subset of the existing predicates that are causing the transition to be spurious. The fewer predicates that are found, the more spurious transitions that are eliminated in one step. The abstract transitions in a spurious counterexample can be examined in any order.

When all SAT instances for the simulation of abstract transitions are satisfiable, it means that none of the abstract transitions is spurious due to the clustering. The immediate conclusion then is that the spurious counterexample is caused

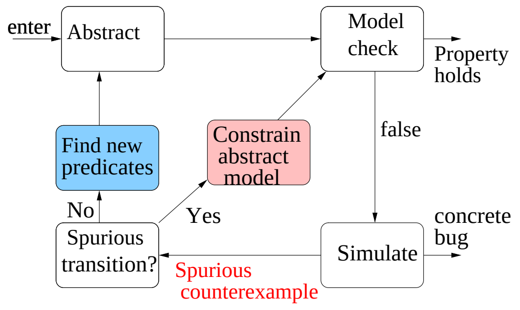


Figure 5.2: Abstraction-refinement loop in this work.

because the predicates used for computing the abstraction were insufficient. For this case, we use the idea of weakest precondition from software model checking [119, 33]. We compute the weakest precondition of the property (or existing predicates) with respect to the transition function given by the circuit to obtain new word-level predicates. We present a technique to avoid the blowup in the size of weakest preconditions when computing the predicates. The use of weakest preconditions provides a good heuristic for finding the predicates for refinement. However, there is no theoretical guarantee that the abstraction refinement loop will make progress with the addition of new predicates[2]. The overall flow of the various techniques described above is shown in Fig. 5.2.

We describe our modeling of a circuit in Section 5.2. Section 5.3 describes SAT-based predicate abstraction with the help of an example. Techniques for clustering the given set of predicates are presented in Section 5.4. We describe techniques for abstraction-refinement in Section 5.5.

[2]In principle this problem can be solved by allowing predicates with universally quantified input variables or indexed predicates [104].

```
module main(clk);
  input clk;
  reg [7:0] x,y;

  initial x = 1;
  initial y = 0;

  always @ (posedge clk) begin
    y <=x;
    if (x<100) x<=y+x;
  end
endmodule
```

Figure 5.3: A Verilog program used as running example.

5.2 Word-Level Transition Functions

Let $\mathcal{R} = \{r_1, \ldots, r_n\}$ denote the set of registers in a given Verilog program. For example, the state of the Verilog program in Fig. 5.3 is defined by the value of the registers x and y, and each of them has a storage capacity of 8 bits. Let S denote the set of states for a given Verilog program.

We treat external inputs like registers without a next-state function. Let $\mathcal{Q} \subseteq \mathcal{R}$ denote the set of registers that are not external inputs, i.e., have a next-state function. We denote the next-state function of a word-level register $r_i \in \mathcal{Q}$ by $f_i(r_1, \ldots, r_n)$, or $f_i(\bar{r})$ using vector notation, where $\bar{r} = \langle r_1, \ldots, r_n \rangle$. We use the word-level next-state functions f_i, to define the transition relation $R(\bar{r}, \bar{r}')$. The transition relation relates the current state $\bar{r} \in S$ to the next state $\bar{r}' \in S$ and is defined as follows:

$$R(\bar{r},\bar{r}') \;:=\; \bigwedge_{r_i \in Q} (r_i' = f_i(\bar{r}))$$

Example 17 Consider the Verilog program in Fig. 5.3. The next-state function for the register `x` is given as follows: if the value of `x` in the current state is less than 100, then the value of `x` in the next state is equal to the sum of current values of `x` and `y`, that is $x+y$. If the value of `x` is greater than or equal to 100, then the value of `x` in the next state remains unchanged. The value of `y` in the next state is equal to the value of `x` in the current state. We use the ternary choice operator $c?g:h$ to denote a function that evaluates to g when the condition c is true, otherwise it evaluates to h. We denote the next-state functions of `x` and `y` as $f_x(x,y)$ and $f_y(x,y)$, respectively, and the transition relation as $R(x,y,x',y')$.

$$\begin{aligned} f_x(x,y) &:= ((x \ < \ 100) \ ? \ (x+y) \ : \ x) \\ f_y(x,y) &:= x \\ R(x,y,x',y') &:= (x' = ((x \ < \ 100) \ ? \ (x+y) \ : \ x)) \wedge (y' = x) \end{aligned}$$

In a *netlist* level representation there is a next-state function for each bit in the registers `x,y`. In contrast, we have a next-state function for the whole registers `x,y` and not for the individual bits of `x,y`. We represent the circuit using *register-level* or *word-level* next-state functions.

Example 18 Consider the Verilog program in Fig. 5.3. We wish to show that

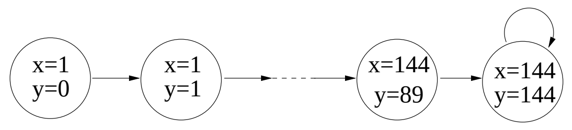


Figure 5.4: The value of x and y in different states

the value of x is always less than 200. Intuitively, the property holds because the value of x follows a sequence starting from 1 to 144. Upon reaching the value 144, the guard in the next-state function for x becomes false, and its value remains unchanged. The values of x and y in each state are shown in Fig. 5.4.

We follow the counterexample guided abstraction refinement (CEGAR) framework in order to prove or disprove a given property. The first step of the CEGAR loop is to obtain an abstraction of the given program.

5.3 Predicate Abstraction

In predicate abstraction [85], the variables of the concrete program are replaced by Boolean variables that correspond to predicates on the variables in the concrete program. These predicates are functions that map a concrete state $\bar{r} \in S$ into a Boolean value. Let $\mathtt{B} = \{\pi_1, \ldots, \pi_k\}$ be the set of predicates. When applying all predicates to a specific concrete state, one obtains a vector of Boolean values, which represents an abstract state $\bar{b}$. We denote this function by $\alpha(\bar{r})$. It maps a concrete state into an abstract state and is therefore called an *abstraction function*.

We perform an existential abstraction [57], i.e., the abstract model can make a transition from an abstract state $\bar{b}$ to $\bar{b}'$ iff there is a transition from $\bar{r}$ to $\bar{r}'$ in the concrete model and $\bar{r}$ is abstracted to $\bar{b}$ and $\bar{r}'$ is abstracted to $\bar{b}'$. We call the abstract machine $\hat{T}$, and we denote the transition relation of $\hat{T}$ by $\hat{R}$. Formally:

$$\hat{R} \ := \{(\bar{b},\bar{b}') \,|\, \exists \bar{r},\bar{r}' \in S: \ \ \alpha(\bar{r}) = \bar{b} \wedge R(\bar{r},\bar{r}') \wedge \alpha(\bar{r}') = \bar{b}'\} \tag{5.1}$$

We refer to a set and its Boolean representation interchangeably. For example, in the above equation $\hat{R}$ denotes a set of abstract transitions. A Boolean (characteristic) function representing this set is denoted as $\hat{R}(\bar{b},\bar{b}')$.

The initial set of states $I(\bar{r})$ is abstracted as follows:

$$\hat{I}(\bar{b}) \ \ := \ \ \exists \bar{r} \in S: \ (\alpha(\bar{r}) = \bar{b}\ ) \wedge I(\bar{r})$$

The abstraction of a safety property $P(\bar{r})$ is defined as follows: for the property to hold on an abstract state $\bar{b}$, the property must hold on all states $\bar{r}$ that are abstracted to $\bar{b}$.

$$\hat{P}(\bar{b}) \ \ := \ \ \forall \bar{r} \in S: (\alpha(\bar{r}) = \bar{b}) \Longrightarrow P(\bar{r})$$

Thus, if $\hat{P}$ holds on all reachable states of the abstract model, P also holds on all reachable states of the concrete model.

The techniques described in this chapter can be used to check any LTL [58] safety property. This is because the spurious counterexamples for LTL safety properties are always finite acyclic paths [65]. Such spurious counterexamples can be removed during the refinement phase (Section 5.5). Predicate abstraction can also be used to verify an arbitrary LTL property, including liveness properties, if the transition relation is total. However, this requires removal of counterexamples containing loops and is left for future research.

SAT-based Predicate Abstraction

In [59], the authors propose to use a SAT solver to compute the abstraction of a sequential ANSI-C program. This approach supports all ANSI-C integer operators, including the bit-vector operators. We use a similar technique for computing the abstraction of Verilog programs.

A symbolic variable b_i is associated with each predicate π_i. Each concrete state $\bar{r} = \langle r_1, \ldots, r_n \rangle$ maps to an abstract state $\bar{b} = \langle b_1, \ldots, b_k \rangle$, where $b_i = \pi_i(\bar{r})$. If the concrete machine makes a transition from state $\bar{r}$ to state $\bar{r}' = \langle r'_1, \ldots, r'_n \rangle$, then the abstract machine makes a transition from state $\bar{b}$ to $\bar{b}' = \langle b'_1, \ldots, b'_k \rangle$, where $b'_i = \pi_i(\bar{r}')$. We refer to $\pi_i(\bar{r})$ as a *current-state* predicate and $\pi_i(\bar{r}')$ as a *next-state* predicate. For example, if $x = y$ denotes a current-state predicate, then its next-state version is $x' = y'$.

The formula that is passed to the SAT solver directly follows from the definition of the abstract transition relation $\hat{R}$ as given in equation 5.1:

$$\hat{R} \quad := \quad \{(\bar{b}, \bar{b}') \mid \exists \bar{r}, \bar{r}' \ : \ \Gamma(\bar{r}, \bar{r}', \bar{b}, \bar{b}')\} \text{ , where} \tag{5.2}$$

$$\Gamma(\bar{r},\bar{r}',\bar{b},\bar{b}') \quad := \quad \bigwedge_{i=1}^{k} b_i = \pi_i(\bar{r}) \wedge R(\bar{r},\bar{r}') \wedge \bigwedge_{i=1}^{k} b'_i = \pi_i(\bar{r}')$$

The set of abstract transitions $\hat{R}$ is computed by transforming $\Gamma(\bar{r},\bar{r}',\bar{b},\bar{b}')$ into conjunctive normal form (CNF) and passing the resulting formula to a SAT solver. Suppose the SAT solver returns $\bar{r},\bar{r}',\bar{b},\bar{b}'$ as the satisfying assignment. We project out all variables but $\bar{b}$ and $\bar{b}'$ from this satisfying assignment to obtain one abstract transition $(\bar{b},\bar{b}')$. Since we want all the abstract transitions, we add a blocking clause to the SAT equation that eliminates all satisfying assignments that assign the same values to $\bar{b}$ and $\bar{b}'$, and re-start the solver. This process is continued until the SAT formula becomes unsatisfiable. The disjunction of abstract transitions obtained gives us the abstract transition relation $\hat{R}$.

Example 19 Let the transition relation $R(x,y,x',y')$ be $x' = y \wedge y' = x$. Let the set of predicates be $\{x = 1, y = 1\}$. The equation for computing $\hat{R}$ is given as follows:

$$\exists x,y,x',y' : (b_1 \Leftrightarrow (x = 1)) \wedge (b_2 \Leftrightarrow (y = 1)) \wedge$$
$$R(x,y,x',y') \wedge (b'_1 \Leftrightarrow (x' = 1)) \wedge (b'_2 \Leftrightarrow (y' = 1))$$

The set of satisfying assignments to the above equation results in $\hat{R}(b_1,b_2,b'_1,b'_2)$ as $((b'_1 \Leftrightarrow b_2) \wedge (b'_2 \Leftrightarrow b_1))$.

The predicates used for abstraction can be arbitrary Boolean expressions allowed by the Verilog syntax. Thus, the predicates can involve operators for concatenation, extraction, etc. For example, `a[3:0]>7`, `ram[{addr,1'b0}]==d[9:2]` are allowed as predicates. Predicates can refer to individual bits in a register. For example, `rg[i]` is a valid predicate, where `rg` is a register and `i` is an index.

The set of abstract initial states can be enumerated using a SAT solver in a similar manner.

5.4 Predicate Clustering

We call the computation of the exact existential abstraction as described in the previous section the *Eager Approach*. A single abstract transition relation is computed using all the available predicates. In the worst case, the number of satisfying assignments generated from Eqn. 5.2 is exponential in the number of predicates. In practice, computing abstractions using the eager approach can be very slow even for a small number of predicates.

The speed of the abstraction step can be increased if we do not aim at the most precise abstract transition relation. That is, we allow our abstraction to be an over-approximation of the abstract transition relation generated by the eager approach. Software predicate abstraction tools abstract the individual statements or basic blocks separately. As only a small number of predicates are typically affected at each statement or basic block, simple heuristics can be used to compute the abstraction quickly. The SLAM toolkit, for example, limits the number of predicates in each theorem prover query. In contrast, each transition in a RT-level circuit consists of simultaneous assignments to all registers. All predicates might change their value in each transition of the circuit. Thus, more sophisticated techniques are needed to compute the predicate abstraction of circuits efficiently.

Our solution to the above problem is as follows: the set of predicates and their next-state versions is clustered into smaller sets of related predicates. We call these sets *clusters*, and denote them by $C_1, \ldots, C_l$, with $C_j \subseteq \{\pi_1, \ldots, \pi_k, \pi'_1, \ldots, \pi'_k\}$. Note that we do not require the clusters to be disjoint, that is, they can have common predicates. We abstract the transition system with respect to each cluster $C_1, \ldots, C_l$. This results in a total of l abstract transition relations $\hat{R}_1, \ldots, \hat{R}_l$, which are conjoined to form $\hat{R}$:

$$\hat{R} \; := \; \bigwedge_{i=1}^{l} \hat{R}_i \tag{5.3}$$

The equation for abstracting the transition system with respect to C_j is given as follows:

$$\hat{R}_j := \exists \bar{r}, \bar{r}' : \bigwedge_{\pi_i \in C_j} b_i = \pi_i(\bar{r}) \;\wedge\; R(\bar{r}, \bar{r}') \;\wedge\; \bigwedge_{\pi'_i \in C_j} b'_i = \pi_i(\bar{r}')$$

The satisfying assignments to the above equation correspond to the abstract transition relation $\hat{R}_j$. The number of satisfying assignments to the above equation is limited by the size of cluster C_j, that is, $2^{|C_j|}$. Clearly, by limiting the size of C_j, we can compute the abstract transition relations much faster as compared to the eager approach.

We refer to the above technique of generating smaller clusters from a given set of predicates, and using these clusters for computing the abstraction $\hat{R}$, as *predi-*

cate clustering.

Proposition 1 *If $\hat{Q}$ denotes the abstract transition relation obtained by using the eager approach (Eqn. 5.2), and $\hat{R}$ denotes the abstract transition relation obtained by predicate clustering (Eqn. 5.3), then $\hat{Q} \Rightarrow \hat{R}$ or $\hat{Q} \subseteq \hat{R}$ using set notation.*

Proof. Let the set of predicates be Pr. $\hat{Q}$ denotes the abstraction with respect to Pr. From Eqn. 5.3, $\hat{R} = \bigwedge_{j=1}^{l} \hat{R}_j$, where $\hat{R}_j$ denotes the abstraction with respect to a cluster C_j and $C_j \subseteq Pr$. The above claim is proved by showing that for all $1 \leq j \leq l$, $\hat{Q} \Rightarrow \hat{R}_j$ or $\hat{Q} \subseteq \hat{R}_j$ using set notation. We will treat $\hat{Q}$ and $\hat{R}_j$ as sets of abstract transitions and show that $\hat{Q} \subseteq \hat{R}_j$. We rewrite the definitions of $\hat{Q}$ and $\hat{R}_j$ as follows:

$$\begin{aligned} \hat{Q} &:= \{(\bar{b}, \bar{b}') \mid \exists \bar{r}, \bar{r}' \ : \ \delta(\bar{r}, \bar{r}', \bar{b}, \bar{b}', Pr)\} \\ \hat{R}_j &:= \{(\bar{b}, \bar{b}') \mid \exists \bar{r}, \bar{r}' \ : \ \delta(\bar{r}, \bar{r}', \bar{b}, \bar{b}', C_j)\} \end{aligned}$$

where $\delta(\bar{r}, \bar{r}', \bar{b}, \bar{b}', Z)$ relates concrete states $\bar{r}, \bar{r}'$, and abstract states $\bar{b}, \bar{b}'$ with respect to a set of predicates Z.

$$\delta(\bar{r}, \bar{r}', \bar{b}, \bar{b}', Z) \ := \ \bigwedge_{\pi_i \in Z} b_i = \pi_i(\bar{r}) \wedge R(\bar{r}, \bar{r}') \wedge \bigwedge_{\pi_i' \in Z} b_i' = \pi_i(\bar{r}')$$

If $Z_2 \subseteq Z_1$ holds, then $\delta(\bar{r}, \bar{r}', \bar{b}, \bar{b}', Z_1)$ is equivalent to $\delta(\bar{r}, \bar{r}', \bar{b}, \bar{b}', Z_2) \wedge \delta(\bar{r}, \bar{r}', \bar{b}, \bar{b}', Z_1 \backslash Z_2)$. Thus, if $\delta(\bar{r}, \bar{r}', \bar{b}, \bar{b}', Z_1)$, then $\delta(\bar{r}, \bar{r}', \bar{b}, \bar{b}', Z_2)$ holds. If an abstract transition $(\bar{a}, \bar{a}') \in$

$\hat{Q}$, then there exist two concrete states $\bar{x}, \bar{x}'$ such that $\delta(\bar{x}, \bar{x}', \bar{a}, \bar{a}', Pr)$ holds. Since $C_j \subseteq Pr$, it follows from the above that $\delta(\bar{x}, \bar{x}', \bar{a}, \bar{a}', C_j)$ holds. Thus, $\exists \bar{r}, \bar{r}' : \delta(\bar{r}, \bar{r}', \bar{a}, \bar{a}', C_j)$ holds and $(\bar{a}, \bar{a}') \in \hat{R}_j$. This shows $\hat{Q} \subseteq \hat{R}_j$. As $\hat{Q} \subseteq \hat{R}_j$ for all $1 \leq j \leq l$ and $\hat{R} = \bigcap_j \hat{R}_j$, it follows that $\hat{Q} \subseteq \hat{R}$. $\square$

We discuss techniques for creating predicate clusters next. Let $var(e)$ denote the set of variables (state elements and inputs) appearing in an expression e. For example, $var(x' + y' < 200)$ is $\{x', y'\}$. If e contains combinational elements, we replace them by their definition in terms of state elements and inputs before computing $var(e)$.

Clarke et al. [55] call two formulas g_1 and g_2 interfering iff $var(g_1) \cap var(g_2) \neq \emptyset$. The authors use the notion of interference to partition a set of formulas into various formula clusters. This technique can be used for clustering the set of predicates as well. However, our early unreported experiments indicate that this results in clusters that are too large. Thus, we make the conditions for keeping the two predicates together stronger, which leads to a smaller number of predicates per cluster. We evaluate two different techniques for creating predicate clusters used in predicate clustering, *cone clustering* and clustering for *lazy abstraction*.

5.4.1 Syntactic Cone Clustering

This technique clusters next state predicates with current state predicates that are related to each other. In order to identify when a next-state predicate is related to a current-state predicate we use *cone of influence* computation [58].

Given a formula g' in terms of next-state variables $\bar{r}'$, the current state variables $\bar{r}$ that affect the value of the variables in $var(g')$ are denoted by $cone(g')$. It is defined as follows: The variables in the next-state functions for the registers mentioned in g' form the cone of g'. Recall that the set of registers is denoted by $\mathcal{Q}$. The next-state function of a particular register $r_i \in \mathcal{Q}$ is given by $f_i(\bar{r})$.

$$cone(g') \quad := \quad \bigcup_{r_i' \in var(g') \,\wedge\, r_i \in \mathcal{Q}} var(f_i(\bar{r}))$$

The value of g' in a given state depends only on the values of variables in $cone(g')$ in the previous state.

Example 20 Let g' be $a' < b'$. Let the next-state functions for a', b' be $x+b$, c, respectively. Here, $var(g') = \{a', b'\}$ and $cone(g') = \{x, b, c\}$. Given the values of x, b, c in a state the value of the predicate $a < b$ in the next state, that is, the value of $a' < b'$ is $x+b < c$. Thus, we would like to keep the current state predicates over variables $\{x, b, c\}$ and the next state predicate $a' < b'$ in the same cluster. This allows the value of the predicate $a' < b'$ to be tracked precisely in the abstract model.

Let the set of predicates and their next-state versions $\{\pi_1, \ldots, \pi_k, \pi_1', \ldots, \pi_k'\}$ be C. The clusters of C are created by the following two steps:

1. The next-state predicates that have identical cone sets are kept in a single cluster. Intuitively, these predicates depend on exactly the same set of vari-

ables from the previous state and hence, are related to each other. That is, if $cone(\pi_i') = cone(\pi_j')$, then π_i' and π_j' are kept in the same cluster. Let $C_1', \ldots, C_l'$ be the clusters of $\{\pi_1', \ldots, \pi_k'\}$ obtained after this step. Since all the predicates in a given cluster C_i' have the same cone, we define $cone(C_i')$ as the cone of any element in C_i'.

2. The final set of clusters is given by $\{C_1, \ldots, C_l\}$. Each C_i contains all the next-state predicates from C_i' and the current-state predicates that mention variables in the cone of C_i'. Formally, C_i is defined as follows:

$$C_i \;:=\; C_i' \cup \{\pi_j \mid var(\pi_j) \subseteq cone(C_i')\}$$

Example 21 Let the transition relation $R(x,y,z,x',y',z')$ be $x' = y \wedge y' = x \wedge z' = x$. Let the set of predicates be $\{x = 2, y = 1, z > 3, x' = 2, y' = 1, z' > 3\}$. The cone sets for the next-state predicates $x' = 2, y' = 1, z' > 3$ are $\{y\}, \{x\}, \{x\}$, respectively. After the first step of the clustering, the clusters are $C_1' := \{x' = 2\}$ and $C_2' := \{y' = 1, z' > 3\}$. Even though $y' = 1$ and $z' > 3$ do not share a common set of variables they are kept in the same cluster, as they have the identical cone set $\{x\}$.

Since $cone(C_1') := \{y\}$ and $cone(C_2') := \{x\}$, the clusters obtained after the second step of the clustering are $C_1 := \{y = 1, x' = 2\}$ and $C_2 := \{x = 2, y' = 1, z' > 3\}$. Observe how the predicates in a given cluster affect each other. For example, in C_2, if $x = 2$ is true, then we know that $y' = 1$ and $z' > 3$ will be false (as y' and z' equal x). If $x = 2$ is false, then $y' = 1$ can be either true or false and $z' > 3$ can be either true or false. However, both $y' = 1$ and $z' > 3$ cannot be true together.

Since cone clustering attempts to keep all related predicates together, the abstractions produced are not much coarser than those produced by the eager approach. However, in general there is no bound on the number of predicates in a given cluster. In the worst case there might be a cluster containing most of the current-state and next-state predicates.

5.4.2 Syntactic Clustering for Lazy Abstraction

The idea of lazy abstraction [88] is to start with a coarse initial abstract model which is refined on-demand as required by spurious counterexamples. Since a coarse abstract model is computed the abstraction step is usually very fast. This prevents the abstraction step from becoming a bottleneck when computing the abstraction of large circuits or when a large number of predicates are available for abstraction.

A completely lazy abstraction corresponds to using no predicate clusters. Thus, the initial abstraction is simply true. We follow a variant of this technique: all current-state predicates that contain the same set of variables are kept in the same cluster. That is, if $var(\pi_i) = var(\pi_j)$, then π_i and π_j are kept in the same cluster. This is useful if the given set of predicates contains many mutually exclusive (or related) predicates such as $x = 1, x = 2, x > 2$. Keeping these predicates in separate clusters results in an abstract model that does not keep track of the relationships between the predicates $x = 1, x = 2, x > 2$. Such an abstract model can contain a large number of spurious abstract states, such as an abstract state in which both $x = 1$ and $x = 2$ are true.

The next-state predicates are not used in the clusters. Thus, the abstraction produced only contains predicate relationships that hold in each abstract state (not between states). If needed the relationships between current-state and next-state predicates is discovered lazily using refinement (Section 5.5).

Example 22 Let the set of current-state predicates be $\{x < 2, x = 1, y = 1, z > 1\}$. The clusters produced for lazy abstraction are $C_1 := \{x < 2, x = 1\}$, $C_2 := \{y = 1\}$, and $C_3 := \{z > 1\}$.

In this example let the next state function of y be equal to x (that is $y' := x$). The predicates involving x and y' are not present together in any cluster. Thus, the abstract model generated using lazy abstraction allows an abstract transition from a state where $x = 1$ to a state where $y \neq 1$. This is a *spurious abstract transition* because the value of $y = 1$ in the next-state must be equal the value of $x = 1$ in the previous state. This fact would be tracked in the most precise abstraction and abstraction computed using cone clustering as the predicates $x = 1, y' = 1$ are kept together in a same cluster.

Once the abstraction of the concrete system is obtained, we model-check it using a model-checker for finite state systems like SMV [3, 11]. If the abstract model satisfies the property, the property also holds on the original, concrete circuit. If the model checking of the abstraction fails, we obtain a counterexample from the model-checker. In order to check if an abstract counterexample corresponds to a concrete counterexample, a *simulation* step is performed. This is done using bounded model checking [42]. If the counterexample cannot be simulated on the concrete model, it is called a *spurious counterexample*. Many spurious counterexamples arise due to predicate clustering. The elimination of spurious counterexamples from the abstract model is described in the next section.

5.5 Abstraction Refinement

When refining the abstract model, we distinguish between two cases of spurious behavior, as done in [63]:

1. **Spurious transitions** are abstract transitions that do not have any corresponding concrete transitions. By definition, spurious transitions cannot appear in the most precise abstraction, which is computed by the eager approach. However, as we noted earlier, computing the most precise abstract model is expensive and thus, we make use of the various predicate clustering techniques which can result in a coarse abstraction. This can result in many spurious transitions.

2. **Spurious prefixes** are prefixes of the abstract counterexample that do not have a corresponding concrete path. This happens when the set of predicates is not rich enough to capture the relevant behaviors of the concrete system, even for the most precise abstraction.

Given a spurious counterexample we first check if any transition in the counterexample is spurious. If a spurious transition is found, it is eliminated from the abstract model by adding a constraint to the abstract model. If no transition in the counterexample is spurious, then new predicates are generated in order to eliminate a spurious prefix in the counterexample. We treat the entire spurious counterexample as a spurious prefix and do not find the shortest spurious prefix.

An abstract counterexample is a sequence of abstract states $\bar{s}(1), \ldots, \bar{s}(l)$, where each abstract state $\bar{s}(j)$ corresponds to a valuation of the k predicates $\pi_1, \ldots, \pi_k$. The value of π_i in a state $\bar{s}$ is denoted by $\bar{s}_i$. Given an abstract state $\bar{s}$, let $\beta(\bar{s})$ denote the conjunction of predicates (or their negation) depending upon their values in $\bar{s}$. For example, let $\bar{s}$ be an abstract state in which the predicate $x < 2$ is true and the predicate $x = y$ is false. Then $\beta(\bar{s}) = x < 2 \wedge \neg(x = y)$.

$$\beta(\bar{s}) := \bigwedge_{i=1}^{k} \pi_i \Leftrightarrow \bar{s}_i$$

We write $\beta(\bar{s}, \bar{r})$ to denote that the variables in $\beta(\bar{s})$ refer to the concrete variables $\bar{r}$.

5.5.1 Detecting and Removing Spurious Transitions

An abstract transition from $\bar{s}$ to $\bar{t}$ is a spurious transition iff there are no concrete states $\bar{r}, \bar{r}'$ such that $\bar{r}$ is abstracted to $\bar{s}$, $\bar{r}'$ is abstracted to $\bar{t}$, and there is a transition from $\bar{r}$ to $\bar{r}'$. Formally, the abstract transition from $\bar{s}$ to $\bar{t}$ is spurious iff the following formula is unsatisfiable:

$$\beta(\bar{s},\bar{r}) \wedge R(\bar{r},\bar{r}') \wedge \beta(\bar{t},\bar{r}')$$

The equation above is transformed into CNF and passed to a SAT solver. If the SAT solver detects the equation to be satisfiable, the abstract transition can be simulated on the concrete model. Otherwise, the abstract transition is spurious. In this case, the spurious transition can be removed from the abstract model by adding a constraint to the abstract model.

When generating the CNF instance for the simulation of the abstract transition $\bar{s}$ to $\bar{t}$, we store the mapping of each predicate π_i, π_i' to the corresponding literal l_i, l_i' in the CNF instance. If the abstract transition is spurious, the CNF instance is unsatisfiable. In this case, we extract an unsatisfiable core [146] from the given CNF instance. An *unsatisfiable core* of a CNF instance is a subset of the original set of clauses that is also unsatisfiable. Current state-of-the-art SAT-solvers are quite effective at producing small unsatisfiable cores, if they exist.

Let us denote the set of current-state predicates whose corresponding CNF literal l_i appears in the unsatisfiable core by X. We have a similar set for the next-state predicates, which we call Y. Intuitively, the predicates in X and Y taken together are sufficient to prove that the abstract transition from $\bar{s}$ to $\bar{t}$ is spurious. All abstract transitions where the predicates in X and Y have the same truth value as given by the states $\bar{s}$ and $\bar{t}$, respectively, are spurious. These spurious transitions are eliminated by adding a constraint to the abstract model. Let b_i and b_i' be the variables used for the predicates π_i and π'_i in the abstract model. The constraint added to the abstract model is as follows:

$$\neg\left(\bigwedge_{\pi_i\in X} b_i \Leftrightarrow \bar{s}_i \;\wedge\; \bigwedge_{\pi'_i\in Y} b'_i \Leftrightarrow \bar{t}_i\right)$$

Proposition 2 *Every abstract transition from $\bar{u}$ to $\bar{v}$ such the predicates in X have the same value in $\bar{u}$ and $\bar{s}$, and the predicates in Y have the same value in $\bar{v}$ and $\bar{t}$, is spurious. The constraint above removes all of these spurious transitions from the abstract model.*

Example 23 Let the set of current-state predicates be $\{x < 2, x = 1, y = 1, z > 1\}$. Consider the abstract transition from $\bar{s} = \{b_1 = 1, b_2 = 1, b_3 = 1, b_4 = 1\}$ to $\bar{t} = \{b'_1 = 0, b'_2 = 0, b'_3 = 0, b'_4 = 0\}$, where b_1, b_2, b_3, and b_4 correspond to the predicates $x < 2$, $x = 1$, $y = 1$, $z > 1$, respectively. Let the next-state function of y be x, i.e., $y' = x$. Observe that in the state $\bar{s}$, $x = 1$. This implies that $y = 1$ in $\bar{t}$ (as $y' = x$). However, b'_3 is false in $\bar{t}$ and thus, the abstract transition from $\bar{s}$ to $\bar{t}$ is spurious. As described in section 5.4.2, the abstract transition from $\bar{s}$ to $\bar{t}$ can arise when using lazy abstraction. This spurious transition can be eliminated by adding the following constraint to the abstract model [69]: $\neg(b_1 \wedge b_2 \wedge b_3 \wedge b_4 \wedge \neg b'_1 \wedge \neg b'_2 \wedge \neg b'_3 \wedge \neg b'_4)$.

However, the constraint above removes just one spurious transition. By examining an unsatisfiable core, we can make the constraint more general, thereby eliminating many spurious transitions at the same time. In this example, the cause of the spurious behavior is $b_2 = 1$, and $b'_3 = 0$. The unsatisfiable core technique described above is capable of discovering this fact. This allows us to eliminate the abstract transition from $\bar{s}$ to $\bar{t}$ and 63 more spurious transitions by adding the following constraint to the abstract model: $\neg(b_2 \wedge \neg b'_3)$. It is very important to remove as many spurious transitions as possible in order to make the CEGAR loop terminate quickly.

Semantic Predicate Clustering

The predicates responsible for making an abstract transition spurious can be treated as a predicate cluster C, which can be used during the abstraction step. Suppose an abstract transition from $\bar{s}$ to $\bar{t}$ is spurious. Let C denote the set of current-state and next-state predicates responsible for this spurious transition as identified by an unsatisfiable core. As described above, the predicates appearing in C are used to remove the spurious transition from $\bar{s}$ to $\bar{t}$. In semantic predicate clustering, C is also added to the existing set of predicate clusters and is used to compute the abstraction (Eqn. 5.3) in the subsequent iterations. Intuitively, the predicates occurring in C are *semantically* related because a particular assignment of truth values to the predicates in C (as given by $\bar{s}$, $\bar{t}$) can make an abstract transition spurious. Thus, by computing all possible relationships between the predicates in C (during abstraction), we remove all abstract transitions that are spurious due to the predicates in C.

Example 24 For the spurious transition in the example above, we obtain $C := \{x = 1, y' = 1\}$. The predicates in C are used to eliminate multiple spurious transitions by adding the constraint $\neg(b_2 \wedge \neg b'_3)$. However, even after adding this constraint the abstract model allows another spurious transition from a state $\bar{u}$ where $\neg(x = 1)$ to a state $\bar{v}$ where $y = 1$ (that is, $y' = 1$). In semantic predicate clustering C is added as a predicate cluster. The abstraction step will discover that $b_2 \Leftrightarrow b'_3$ using C. Thus, the spurious transition from $\bar{u}$ to $\bar{v}$ cannot arise.

5.5.2 Detecting and Removing Spurious Prefixes

An abstract counterexample $\bar{s}(0), \ldots, \bar{s}(l)$ of length l is a spurious prefix iff there is no concrete execution of l transitions such that at each step the concrete state is consistent with the corresponding abstract state. More formally, let $\bar{r}_0, \ldots, \bar{r}_l$ denote the concrete state variables at each of the $l+1$ states. The initial state of the concrete system is denoted as $I(\bar{r}_0)$.

The abstract counterexample $\bar{s}(0), \ldots, \bar{s}(l)$ is a spurious prefix iff the following formula is unsatisfiable:

$$I(\bar{r}_0) \;\wedge\; \bigwedge_{i=0}^{l-1} R(\bar{r}_i, \bar{r}_{i+1}) \;\wedge\; \bigwedge_{i=0}^{l} \beta(\bar{s}(i), \bar{r}_i)$$

The above formula is unsatisfiable iff there is no sequence of concrete states $\bar{r}_0, \ldots, \bar{r}_l$ such that $\bar{r}_0$ is an initial state, there is a transition from $\bar{r}_i$ to $\bar{r}_{i+1}$ for $0 \leq i < l$, and the predicate values in each concrete state $\bar{r}_j$ exactly match the predicate values given by the abstract state $\bar{s}(j)$ for $0 \leq j \leq l$.

In [63], the elimination of spurious prefixes is done by adding a bit-level predicate. This predicate is called a *separating* predicate and is computed by using a SAT based conflict dependency analysis. In contrast, we make use of weakest preconditions as done in software verification. We generate new word-level predicates from the weakest pre-condition of the given property with respect to the transition function given by the RT-level circuit.

Weakest pre-conditions: In software verification, the weakest pre-condition $wp(st, \gamma)$ of a predicate γ is usually defined with respect to a statement *st* (e.g., an assignment). It is the weakest formula whose truth before the execution of *st* entails the truth of γ after *st* terminates. In case of hardware, each state transition can be viewed as a statement where the registers are assigned values according to their next-state functions.

Recall that the set of registers that have a next-state function is denoted by $\mathcal{Q}$. External inputs do not appear in this set. The next-state function for register $r_i \in \mathcal{Q}$ is given by $f_i(\bar{r})$. We use $\bar{f}$ to denote the vector of the next-state functions for the registers in $\mathcal{Q}$. For any expression e, the expression $e[\bar{x}/\bar{g}]$ denotes the simultaneous substitution of each variable x_i in e by an expression g_i from $\bar{g}$.

The weakest precondition of the property $\gamma(\bar{r})$ with respect to one concrete transition is defined as follows:

$$wp_1(\bar{f}, \gamma(\bar{r})) \quad := \quad \gamma(\bar{r})\ [\bar{r}/\bar{f}]$$

The weakest precondition with respect to i consecutive concrete transitions is defined inductively as follows:

$$wp_i(\bar{f},\, \gamma(\bar{r})) \;\; := \;\; wp_1(\bar{f},\, wp_{i-1}(\bar{f},\, \gamma(\bar{r}))) \;\; (i > 1)$$

In order to refine a spurious prefix of length $l > 0$, we compute $wp_i(\bar{f}, \tau)$ for each $1 \leq i \leq l$, where τ is the safety property we are interested in checking. Intuitively, τ holds after i transitions iff $wp_i(\bar{f}, \tau)$ holds before i transitions. Refinement corresponds to adding the Boolean expressions occurring in each $wp_i(\bar{f}, \tau)$ to the existing set of predicates. The refinement procedure is not guaranteed to make progress.

In case of circuits, the weakest pre-condition is always computed with respect to the same transition function $\bar{f}$ and thus, we may omit it as an argument in $wp_i(\bar{f}, \gamma)$.

Example 25 Let the property be $x < 200$. Let the next state functions for the registers x and y be $((x < 100)?(x+y) : x)$ and x, respectively. Suppose we obtain a spurious prefix of length 1. The weakest pre-condition is computed as follows:

$$wp_1(x < 200) \;\; := \;\; (((x \; < \; 100) \; ? \; (x+y) \; : \; x \;) < \; 200)$$

We add the Boolean conditions occurring in wp_1 to our set of predicates. Thus, we add $x < 100$ and $(((x \; < \; 100) \; ? \; (x+y) \; : \; x \;) < \; 200)$ as the new predicates.

Simplifying the Weakest Pre-conditions

When the spurious prefix is long, the weakest precondition computation becomes expensive and the predicates generated can become very complex (see wp_1 above). This adversely affects the abstraction refinement loop. In software verification, this problem is solved by computing the weakest precondition with respect to the statements appearing in the spurious trace only. This is not directly applicable to a synchronous circuit because the statements occurring in the spurious trace correspond to the next state functions. The next-state functions usually contain many conditional statements. Thus, simply substituting the next-state functions as done above leads to a blowup in the size of weakest pre-conditions.

Instead, we apply a syntactic simplification to the weakest preconditions at each step. The simplification uses data from the abstract error trace. We exploit the fact that many of the control flow guards in the Verilog code are also present in the current set of predicates. The abstract trace assigns truth values to these predicates in each abstract state. In order to simplify the weakest pre-conditions, we substitute the guards in the weakest pre-conditions with their truth values. Furthermore, we only add the atomic Boolean expressions occurring in the weakest pre-condition as the new predicates.

In order to formalize the simplification of weakest pre-conditions we define a helper function *simplify* in Algorithm 5.1. Let the current set of predicates be $\{\pi_1, \ldots, \pi_k\}$. *simplify* takes as input a Boolean formula $g(\bar{r})$ (written as g for short) and an abstract state $\bar{t}$. It replaces all the occurrences of $\{\pi_1, \ldots, \pi_k\}$ in g by their truth values in the state $\bar{t}$.

Algorithm 5.1 Simplification of a Boolean formula using the predicate valuations in an abstract state.

```
Input: Boolean expression $g$
Input: An abstract state $\bar{t}$ assigning values to predicates $\{\pi_1, \ldots, \pi_k\}$
Output: $g$ is simplified (modified in-place)
1: for all operands $h$ in $g$ do
2:     $simplify(h, \bar{t})$ {recursive simplification}
3: end for
4: Remove constant conditionals from $g$ {E.g., replace $(0?x : y)$ by $y$}
5: if $\exists \pi_j.(\pi_j = g)$ {syntactic equality of expressions} then
6:     $g \Leftarrow \bar{t}_j$ {replace $g$ by value of $\pi_j$ in $\bar{t}$}
7: end if
```

Example 26 Suppose our current set of predicates is $\{x < 2, x < 1\}$. Let $\bar{t}$ be an abstract state in which $x < 2$ is true and $x < 1$ is true. Let $g(x,y)$ be the formula $(((x\ <\ 1)\ ?\ (x+y)\ :\ x\ ) <\ 2)$. After calling *simplify* with g and t as arguments g becomes:

$$((true?\ (x+y)\ :\ x\ ) <\ 2) \quad = \quad x+y < 2$$

Let $h(x,y)$ be the formula $x < 3$. After calling *simplify* with h and t as arguments h remains equal to $x < 3$.

Simplified weakest pre-conditions Let the spurious prefix be $\bar{t}(0), \ldots, \bar{t}(l)$ with $l \geq 1$ and the property be γ. The weakest precondition wp_i is a formula that should hold before i concrete transitions for γ to hold after i transitions. That is, γ holds after l transitions starting from the initial state iff wp_l holds in the initial state.

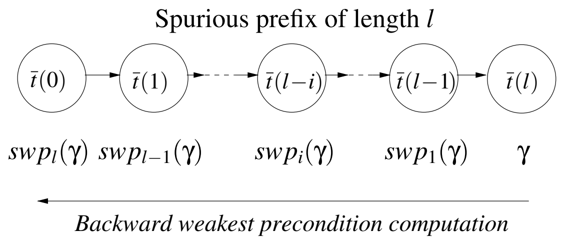


Figure 5.5: Simplified weakest precondition computation for a spurious prefix.

As motivated earlier we want to simplify wp_i using the predicate values from the spurious prefix. We denote the *simplified weakest precondition (swp)* for i steps by swp_i. The abstract state $\bar{t}(l-i)$ provides the truth values of the predicates just before the i transitions leading to the end of spurious prefix. Thus, $swp_i(\gamma)$ is simplified using the predicate values from the abstract state $\bar{t}(l-i)$. Fig. 5.5 shows the correspondence between abstract states and swp_i. Formally, swp_i is defined as follows (wp_1 was defined earlier and l is the length of spurious prefix):

$$\begin{aligned} swp_1(\gamma) &:= \mathit{simplify}(wp_1(\gamma),\, \bar{t}(l-1)) \\ swp_i(\gamma) &:= \mathit{simplify}(wp_1(swp_{i-1}(\gamma)),\, \bar{t}(l-i)) \quad (1 < i \leq l) \end{aligned}$$

The new set of predicates for refinement is obtained from $swp_1, \ldots, swp_l$. This is done by taking only the atomic predicates occurring in the simplified weakest pre-condition.

The predicates in the simplified weakest precondition of the given property are not always sufficient to ensure that the spurious prefix is eliminated from the abstract model. We identify a subset of the existing predicates such that computing the weakest pre-condition of these predicates is likely to remove the spurious prefix. As in [94], this is done by examining the unsatisfiable core of the SAT instance used for simulating the prefix. This approach identifies a subset of the existing predicates that is responsible for the spurious behavior. If a copy of predicate p in cycle k appears in the unsatisfiable core, we compute the weakest precondition of p for k steps ($k \leq l$). In addition we compute the weakest precondition for each predicate used during the simplification (Algorithm 5.1, Line 5).

5.6 Experimental Results

The experiments are performed on a 1.86 GHz Intel Xeon (R) machine with 4 GB of memory running Linux. The techniques described in this chapter have been implemented in a tool called `VCEGAR` [23]. Our implementation is available for experimentation by other researchers. We use the MiniSat (version 1.14) SAT solver [8] as our decision procedure. The abstractions are model checked using a publicly available version of the Cadence SMV model checker [3]. We perform two sets of experiments:

1. We compare the performance of `VCEGAR` with the performance of k-induction [130] and interpolation [112] verification techniques implemented in EBMC [5]. The implementation of interpolation in EBMC uses the ideas from [112, 131] but does not incorporate the optimizations described in [112][3]. The results are reported in Section 5.6.1.

[3]The publically available version of Cadence SMV does not include the interpolation options.

2. We compare three different predicate clustering algorithms: syntactic cone clustering, clustering for lazy abstraction described in Section 5.4, and semantic predicate clustering (Section 5.5.1). These results are reported in Section 5.6.2.

In all our experiments we compute the initial abstraction using the atomic predicates appearing in the property. The remaining predicates are discovered automatically using refinement.

5.6.1 Comparison with Other Verification Techniques

The results are summarized in Table 5.6.1. The column "Latches" contains the total number of latches in the design. The columns marked with "Predicate Abstraction" contain the results of applying the techniques discussed in this chapter. The "Time", "Abs", "MC", and "Ref" columns contain the total time, followed by the time taken by abstraction, model checking, and refinement including simulation. The time spent before the start of the CEGAR loop is given by Time-(Abs+MC+Ref). We use lazy abstraction and rely on refinement to do most of the work in these benchmarks. The "P" column contains the final number of predicates. The "I" column gives two numbers separated by a slash: 1) Number of refinement steps in which spurious transitions are removed, and 2) number of refinement steps in which new predicates are added. The sum of these two numbers is the total number of refinement iterations.

The results of running EBMC with k-induction options are given in the "EBMC-

Bench-mark	Latches	Predicate Abstraction						EBMC-K	EBMC-I
		Time	Abs	MC	Ref	P	I	Time	Time
USB1	545	42	1	2	29	17	62/0	0.60 (1)	2 (1/5)
USB2	545	599	47	147	386	116	146/22	43 (14)	30 (14/20)
USB3	545	446	46	73	317	114	123/20	- (80)	14 (4/18)
ETH0	359	44	2	3	30	21	55/0	- (74)	1213 (19/55)
ETH1	359	127	8	8	102	93	49/2	- (87)	3905 (36/87)
ETH2	359	161	8	16	127	94	109/2	- (83)	-
ETH3	359	204	8	20	166	96	123/2	- (76)	-
ETH4	359	15	0	0	5	4	9/0	- (146)	6 (4/11)
ETH5	359	104	8	6	79	94	54/2	- (82)	-
ETH6	359	161	4	7	140	63	71/5	- (83)	-
ETH7	359	497	6	206	275	77	86/5	- (86)	939 (32/67)
ETH8	359	230	6	33	181	78	47/4	- (86)	733 (29/68)
ETH9	359	222	7	15	190	84	71/5	- (85)	1305 (30/78)
ETH10	359	123	8	6	99	94	46/1	- (82)	-
ETH11	359	11	0	0	1	2	2/0	1 (1)	1 (4/3)
M2KB	16427	5	0	0	5	3	2/0	4.2 (1)	18 (1/2)
M8KB	65694	28	0	0	28	3	2/0	38.4 (1)	293 (1/2)
M16KB	131117	34	0	0	34	3	2/0	44 (1)	308 (1/2)
N2KB	16427	93	0	0	93	11	9/0	39 (1)	452 (1/2)
N8KB	65694	490	0	0	490	11	9/0	550 (1)	-
N16KB	131117	790	0	0	789	11	9/0	679 (1)	-
AR200	400	1	0	0	1	3	3/2	0.1 (2)	0.6 (2/8)
AR3000	6000	12	0	0	12	3	3/2	1.1 (2)	20 (2/10)
AR4000	8000	17	0	0	16	3	3/2	1.5 (2)	28 (2/10)

Table 5.1: Experimental results: All runtimes are in seconds (rounded to nearest integer). A dash "-" indicates a timeout of 2 hours.

K" column. We report the total runtime followed by the k-induction bound at which the property is (dis)proved or a timeout is reached. The "EBMC-I" column gives the runtimes of EBMC with the interpolation options followed by (a) the BMC bound at which the property is (dis)proved and (b) the total number of number of iterations (see FiniteRun procedure in [112]). The interpolation options given to EBMC are `--interpolation --stop-minimize --stop-induction` and optionally `--no-netlist` is provided if it im-

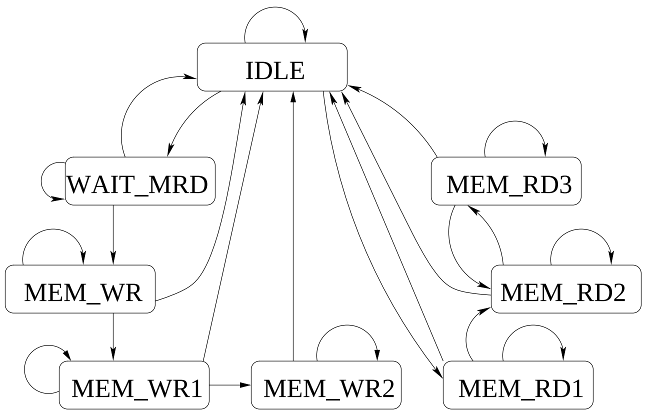


Figure 5.6: State machine for the DMA in the USB 2 .0 Function core.

proves the runtime.

Benchmarks: The USB benchmark was used for experimental evaluation of the EverLost tool [72]. It is derived from a USB 2.0 Function core [12] and contains approximately 4000 lines of RTL Verilog. We checked three properties. The first property USB1 checks that the implementation of the internal DMA module simulates the state transition diagram shown in Fig. 5.6. The property holds and all the predicates required for the proof are present in the property itself. The second property USB2 encodes the following: if the `abort` signal is true in any state of Fig. 5.6, then the next state will be `IDLE`. This property does not hold because the transition from the `MEM_WR2` state to the `IDLE` state is not guaranteed by the `abort` signal. The third property USB3 excludes the state `MEM_WR2` from the USB2 property. This property holds on the design. The properties USB2 and USB3 contain three and four atomic predicates, respectively. The remaining

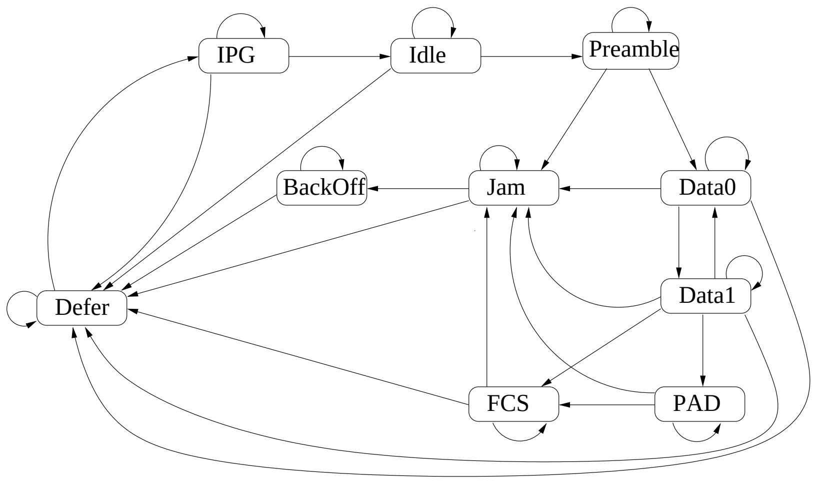


Figure 5.7: State machine for the Transmit module in the Ethernet MAC.

predicates are discovered through refinement.

The ETH benchmark was also used in [72]. It is the design of a 10/100 Mbps Ethernet MAC [12] and contains approximately 5000 lines of RTL Verilog. The transmit module of the design contains a state machine with ten states (see Fig. 5.7). The property ETH0 checks that the implementation obeys the state machine description given in Fig. 5.7. All the predicates required for proving the property are present in the property itself. The property ETH1 checks the outgoing transitions from the state `BackOff`. The property ETH2 checks the outgoing transitions from the state `Jam`. The properties ETH3 to ETH11 are similar and check the outgoing transitions from the remaining states. All properties ETH1 to ETH11 hold on the design. When checking the properties ETH1 to ETH11 most of the predicates are discovered through refinement.

The ICRAM benchmark is taken from the Instruction Cache RAM unit of the

Sun PicoJava II microprocessor [20]. It maintains a RAM of size 16KB (organized as 2048 entries of 64 bits each). If the writing signal `wen0` is enabled the value of data input (`din`) is written to the lower 32 bits of the location addressed by the input address (`addr`). Otherwise, if the writing signal `wen1` is enabled, the value `din` is written to the higher 32 bits of the location addressed by `addr`. This functionality of the ICRAM is encoded in form of eight safety properties using the current-state and next-state of the variables. We use `P.x` to denote the value of a register or input `x` in the previous state. Each property compares eight bits in `P.din` and corresponding bits in ICRAM. A sample property is given below:
`P.wen0→(ram[{P.addr,3'b001}]=P.din[23:16])`

The above property depends on the contents of the RAM. We verified the above property by varying the size of RAM from 2KB to 16KB. These benchmarks start with a prefix "M" in Table 5.6.1. We also combined all the eight properties for the ICRAM benchmark into a single property. These benchmarks start with a prefix "N" in Table 5.6.1. For both "M" and "N" benchmarks the property is proved using only the predicates occurring in the property. No new predicates are discovered.

The benchmarks with names starting with "AR" perform arithmetic operations on two registers x and y as shown in Fig. 5.3. We verify the invariant $x < 200$. In the ARi benchmark the size of both x, y is i and total number of latches is $2 \times i$. As described in the previous section, this property is proved using the predicates

$x < 200, x < 100, x+y < 200$. The predicate $x < 200$ is obtained from the property and the predicates $x < 100, x+y < 200$ are discovered using refinement.

VCEGAR is able to solve all benchmarks reported in Table 5.6.1, while EBMC-K and EBMC-I timeout on 12, 7 problems, respectively. Due to the use of lazy abstraction in VCEGAR the refinement step (simulation of abstract transitions/-counterexamples) takes more than 50% of the runtime.

When using predicate abstraction, the size of the abstract model can remain constant even when the number of latches is increased. This is because for certain properties, the number of word-level predicates needed for the proof does not grow as the width of the registers is increased. This trend is visible in the M*, N*, and AR* benchmarks. Thus, the model checking (MC) time is similar across these benchmarks.

5.6.2 Comparing Predicate Clustering Techniques

We report the performance of the CEGAR loop using three different predicate clustering techniques described in Section 5.4 and Section 5.5.1. The benchmark characteristics are given in Table 5.2. We report the number of lines of code, the total number of latches, the total number of Verilog combinational elements and inputs ("CE+I" column), and the total number of properties checked for each benchmark. The benchmarks USB 2.0 and Ethernet MAC were described in the previous section. Other benchmarks are taken from the Texas97 and VIS [139] benchmark suites.

The results are summarized in Table 5.3. The columns labeled with "Cone"

Benchmark	Lines	Latches	CE+I	Properties
`mpeg`	1215	599	234	2
`SDLX`	898	41	40	1
`Miim`	841	83	173	1
`ethernet (enet)`	610	91	156	2
`itc99-b12 (b12)`	558	151	723	1
`usb-phy (uphy)`	1054	44	25	1
`USB 2.0 (USB)`	4000	545	1686	3
`Ethernet MAC (ETH)`	5000	359	2363	3

Table 5.2: Benchmark characteristics

contain the results of using syntactic cone clustering in the CEGAR loop. The performance of the CEGAR loop when using clustering for lazy abstraction is summarized in the columns labeled with "Lazy". The "Semantic" column presents the results of using semantic predicate clustering (Section 5.5.1).

For each predicate clustering technique, the "Total", "Abs", "MC", and "Ref" columns contain the total verification time, followed by the time taken by abstraction, model checking, and refinement including simulation. The "Preds" column contains two numbers separated by a slash: 1) The total number of predicates in the last iteration of the CEGAR loop. This includes only the current-state predicates. 2) The maximum number of predicates present in any predicate cluster generated by the predicate clustering technique. The number of refinement iterations is reported in the "I" column. The "Res" column contains T (true) if the property holds, else it contains F (false), followed by the length of the counterexample. In these benchmarks (expect USB1, ETH0) most of the predicates are discovered automatically during refinement phase. Below, we compare the three instantiations of the CEGAR loop, which are "Cone", "Lazy", and "Semantic".

"Cone" versus "Lazy": The "Lazy" technique is able to handle all benchmarks within the timeout, and thus, it is more robust than the "Cone" technique (which timeouts on five problems). When using the "Cone" technique, the SAT-based abstraction becomes the bottleneck. Model checking of abstract models also becomes expensive (see `Miim` row). This happens because the abstract models created in the "Cone" technique are more detailed and thus harder. However, the properties can usually be checked using coarse (less precise) abstractions created by the "Lazy" technique.

"Semantic" versus "Lazy": In the "Semantic" technique (Section 5.5.1), new predicate clusters are generated as follows: When a spurious transition is found, we identify a set of predicates responsible for spurious behavior. These predicates are treated as a new predicate cluster. In our experiments this cluster is used during abstraction computation only if it has ≤ 6 predicates. In addition, we use the same predicate clusters as for the "Lazy" technique.

The "Semantic" technique consistently requires fewer refinement iterations than the "Lazy" technique. This shows that computing all possible abstract transitions for the predicates responsible for a spurious transition also rules out other spurious transitions. The runtime of both techniques is comparable.

The abstraction computation or abstraction model checking can become a bottleneck when using the "Cone" technique, while a large number of refinement iterations can hurt the performance when using the "Lazy" technique. The "Semantic" technique tries to balance the bottlenecks of both "Cone" and "Lazy" techniques, and thus, seems to be the most scalable.

5.7 Chapter Summary

We apply the idea of predicate abstraction from software verification to verify hardware designs at a higher level of abstraction. We show how to reduce the abstraction computation overhead in presence of a large number of predicates. This is done by dividing the set of predicates into clusters of related predicates and the abstraction is computed separately for each cluster. In lazy abstraction the expensive task of program abstraction is deferred until a spurious counterexample is found. We show the benefit of lazy abstraction in the context of hardware verification.

We use unsatisfiable cores in order to eliminate multiple spurious transitions. The spurious trace may also be caused by insufficient predicates. We use weakest preconditions to compute new predicates. Our experimental results show that this technique is effective in discovering new word-level predicates for refinement.

Bench-	Cone						Lazy						Res
mark	Time	Abs	MC	Ref	Preds	I	Time	Abs	MC	Ref	Preds	I	
mpeg1	44	25	1	18	31/33	7	41	3	1	37	31/22	24	T
mpeg2	51	26	1	23	31/32	9	47	4	1	43	30/22	26	T
SDLX	8	4	1	2	32/13	23	14	1	5	8	32/6	83	T
Miim	170	49	119	2	23/19	19	8	1	2	6	23/4	55	T
enet1	-	-	-	-	-	-	45	2	20	22	48/4	129	F(6)
enet2	38	6	5	27	37/11	36	69	2	20	47	37/3	117	T
b12	310	181	69	57	50/24	29	132	3	24	103	38/8	148	F(14)
uphy	13	1	3	8	42/18	29	24	0	10	13	42/7	100	F(36)
USB1	12	1	0	0	17/17	0	42	1	2	29	17/8	62	T
USB2	-	-	-	-	-	-	599	47	147	386	116/15	168	F(14)
USB3	-	-	-	-	-	-	446	46	73	317	114/15	143	T
ETH0	49	15	4	19	21/11	31	44	2	3	30	21/0	55	T
ETH1	-	-	-	-	-	-	127	8	8	102	93/0	51	T
ETH2	-	-	-	-	-	-	161	8	16	127	94/0	111	T

Bench-	Semantic					
mark	Time	Abs	MC	Ref	Preds	I
mpeg1	46	10	1	35	30/22	22
mpeg2	54	11	1	43	31/22	24
SDLX	13	3	3	6	32/6	64
Miim	8	2	1	4	23/6	40
enet1	45	6	17	21	48/6	121
enet2	66	6	19	41	37/6	99
b12	131	17	13	98	48/8	94
uphy	23	1	10	11	42/7	87
USB1	51	19	1	20	17/8	40
USB2	547	109	87	333	116/15	139
USB3	459	97	70	282	114/15	120
ETH0	57	14	3	30	21/6	53
ETH1	177	48	13	107	93/6	54
ETH2	172	48	14	100	94/6	95

Table 5.3: Comparing three CEGAR loops each employing a different predicate clustering method. All times are reported in seconds (rounded to nearest integer). A dash "-" indicates a timeout of 2 hours

Chapter 6

Interpolation for Subsets of Integer Linear Arithmetic

The use of Craig interpolants has enabled the development of powerful hardware and software model checking techniques [112, 89, 99]. Efficient algorithms are known for computing interpolants in rational and real linear arithmetic. In this chapter we present efficient interpolation algorithms for subsets of integer linear arithmetic or $LA(\mathbb{Z})$.

Informally, a linear equation where all variables are integer variables is said to be a *linear diophantine equation (LDE)*. A *linear modular equation (LME)* or a *linear congruence* over integer variables is a type of linear equation that expresses divisibility relationships. A *system* of LDEs (LMEs) denotes a conjunction of LDEs (LMEs). Both LDEs and LMEs arise naturally in program verification when modeling assignments and conditional statements as logical formulas.

These subsets of $LA(\mathbb{Z})$ are also known to be tractable, that is, polynomial time algorithms are known for deciding systems of LDEs and LMEs. We study the interpolation problem for LDEs and LMEs. Our contributions are summarized below.

6.1 Contributions

Given formulas F, G such that $F \wedge G$ is unsatisfiable. An interpolant for the pair (F, G) is a formula $I(F, G)$ with the following properties: (i) F implies $I(F, G)$, (ii) $I(F, G) \wedge G$ is unsatisfiable, and (iii) $I(F, G)$ refers only to the common variables of F and G. This thesis presents the following new results.

- Let F, G denote systems of LDEs. We show that $I(F, G)$ can be obtained in polynomial time by using a proof of unsatisfiability of $F \wedge G$. The interpolant can be either a LDE or a LME. This is because in some cases there is no $I(F, G)$ that is a LDE. In these cases, however, there is always an $I(F, G)$ in the form of a LME. (Section 6.4)

- Let F, G denote systems of LMEs. We obtain $I(F, G)$ in polynomial time by using a proof of unsatisfiability of $F \wedge G$. We can ensure that $I(F, G)$ is a LME. (Section 6.5)

- Let S denote an unsatisfiable system of LDEs. The proof of unsatisfiability of S can be obtained in polynomial time by using the *Hermite Normal Form* of S (represented in matrix form). A system of LMEs R can be reduced to an equi-satisfiable system of LDEs R'. The proof of unsatisfiability for R is easily obtained from the proof of unsatisfiability of R'. (Section 6.6)

- Let S denote a system of LDEs. We show that if S has an integral solution, then every LDE that is implied by S, can be obtained by a linear combination of equations in S. We show that S is *convex* [120], that is, if S implies a disjunction of LDEs, then it implies one of the equations in the disjunction. In contrast, conjunctions of atomic formulas in $LA(\mathbb{Z})$ are not convex due to inequalities [120]. These results help in efficiently dealing with *linear diophantine disequations (LDDs)*. (Section 6.7)

- Let $S = S_1 \wedge S_2$, where S_1 is a system of LDEs, while S_2 is a system of LDDs. We say that S is a system of LDEs+LDDs. We show that S has no integral solution if and only if $S_1 \wedge S_2$ has no rational solution or S_1 has no integral solution. This gives a polynomial time decision procedure for checking if S has an integral solution. If S has no integral solution, then the proof of unsatisfiability of S can be obtained in polynomial time. (Section 6.7)

- Let F, G denote systems of LDEs+LDDs. We show $I(F, G)$ can be obtained in polynomial time. The interpolant can be an LDE, an LDD, or an LME. (Section 6.7)

- We show the utility of our interpolation algorithms in counterexample guided abstraction refinement (CEGAR) based verification [56]. Our interpolation algorithm is effective at discovering *modular/divisibility predicates*, such as $3x + y + 2z \equiv 1 \ (mod\ 4)$, from spurious counterexamples. This has allowed us to verify programs that cannot be verified by existing hardware and software model checkers. (Section 6.8)

Polynomial time algorithms are known for solving (deciding) a system of LDEs [129, 44] and LMEs (by reduction to LDEs) over integers. We do not give any new algorithms for solving a system of LDEs or LMEs. Instead we focus on obtaining proofs of unsatisfiability and interpolants for systems of LDEs, LMEs, and LDEs+LDDs. We only consider conjunctions of LDEs, LMEs, and LDEs+LDDs. Interpolants for any (unsatisfiable) Boolean combination of LDEs can also be obtained by calling the interpolation algorithm for conjunctions of LDEs+LDDs multiple times in a satisfiability modulo theory (SMT) framework [54]. However, computing interpolants for Boolean combinations of LMEs is difficult. This is due to linear modular disequations (LMDs). We can show that even the decision problem for conjunctions of LMDs is NP-hard.

All proofs are present in the appendix D.

6.2 Related Work

It is known that Presburger arithmetic (PA) augmented with divisibility predicates allows quantifier elimination [125]. Kapur et al. [100] show that a recursively enumerable theory allows quantifier-free interpolants if and only if it allows quantifier elimination. The systems of LDEs, LMEs, LDEs+LDDs are subsets of PA.

Thus, the existence of quantifier-free interpolants for these systems follows from [100]. However, quantifier elimination for PA has an exponential complexity and does not immediately yield efficient algorithms for computing interpolants. We give polynomial time algorithms for computing proofs of unsatisfiability and interpolants for systems (conjunctions) of LDEs, LMEs, LDEs+LDDs.

Let S_1, S_2 denote conjunctions of atomic formulas in $LA(\mathbb{Z})$. Suppose $S_1 \wedge S_2$ is unsatisfiable. Pudlak [126] shows how to compute an interpolant for (S_1, S_2) by using a *cutting-plane* (CP) proof of unsatisfiability. The CP proof system is a sound and complete way of proving unsatisfiability of conjunctions of atomic formulas in $LA(\mathbb{Z})$. However, a CP proof for a formula can be exponential in the size of the formula. Pudlak does not provide any guarantee on the size of CP proofs for a system of LDEs or LMEs. Our results show that polynomially sized proofs of unsatisfiability and interpolants can be obtained for systems of LDEs, LMEs and LDEs+LDDs.

McMillan [113] shows how to compute interpolants in the combined theory of rational linear arithmetic $LA(\mathbb{Q})$ and equality with uninterpreted functions $\mathcal{EUF}$ by using proofs of unsatisfiability. Rybalchenko and Sofronie-Stokkermans [128] show how to compute interpolants in combined $LA(\mathbb{Q})$, $\mathcal{EUF}$ and real linear arithmetic $LA(\mathbb{R})$ by using linear programming solvers in a black-box fashion. The key idea in [128] is to use an extension of Farkas lemma [129] to reduce the interpolation problem to constraint solving in $LA(\mathbb{Q})$ and $LA(\mathbb{R})$. Cimatti et al. [54] show how to compute interpolants in a satisfiability modulo theory (SMT) framework for $LA(\mathbb{Q})$, rational difference logic fragment and $\mathcal{EUF}$. By making use of state-of-the-art SMT algorithms [74] they obtain significant improvements over existing interpolation tools for $LA(\mathbb{Q})$ and $\mathcal{EUF}$. Yorsh and Musuvathi [144] give a Nelson-Oppen [120] style method for generating interpolants in a combined theory by using the interpolation procedures for individual theories. Kroening and Weissenbacher [101] show how a bit-level proof can be lifted to a word-level proof of unsatisfiability (and interpolants) for equality logic.

To the best of our knowledge the work in [113, 144, 128, 101, 54] is not complete for computing interpolants in $LA(\mathbb{Z})$ or its subsets such as LDEs, LMEs, LDEs+LDDs. That is, the work in [113, 144, 128, 101, 54] cannot compute interpolants for formulas that are satisfiable over rationals but unsatisfiable over integers. Such formulas can arise in both hardware and software verification. We give sound and complete polynomial time algorithms for computing interpolants for conjunctions of LDEs, LMEs, LDEs+LDDs. Efficient interpolation algorithms for LDEs, LMEs, LDEs+LDDs are also crucial in order to develop practical interpolating theorem provers for $LA(\mathbb{Z})$ and bit-vector arithmetic [68, 38, 32, 81, 107, 49, 82, 48].

6.3 Notation and Preliminaries

We use capital letters $A, B, C, X, Y, Z, \ldots$ to denote matrices and formulas. A matrix M is *integral (rational)* iff all elements of M are integers (rationals). For a matrix M with m rows and n columns we say that the size of M is $m \times n$. A *row vector* is a matrix with a single row. A *column vector* is a matrix with a single column.

We sometimes identify a matrix M of size 1×1 by its only element. If A, B are matrices, then AB denotes matrix multiplication. We assume that all matrix operations are well defined. For example, when we write AB without specifying the sizes of matrices A, B, it is assumed that the number of columns in A equals the number of rows in B.

For any rational numbers α and β, $\alpha | \beta$ if and only if, α divides β, that is, if and only if $\beta = \lambda\alpha$ for some integer λ. We say that α is equivalent to β *modulo* γ written as $\alpha \equiv \beta \ (mod\ \gamma)$ if and only if $\gamma | (\alpha - \beta)$. We say γ is the *modulus* of the equation $\alpha \equiv \beta \ (mod\ \gamma)$. We allow α, β, γ to be rational numbers. If $\alpha_1, \ldots, \alpha_n$ are rational numbers, not all equal to 0, then the largest rational number γ dividing each of $\alpha_1, \ldots, \alpha_n$ exists [129], and is called the *greatest common divisor*, or *gcd* of $\alpha_1, \ldots, \alpha_n$ denoted by $gcd(\alpha_1, \ldots, \alpha_n)$. We assume that gcd is always positive.

Basic Properties of Modular Arithmetic: Let a, b, c, d, m be rational numbers.

P1. $a \equiv a \ (mod\ m)$ (reflexivity).

P2. $a \equiv b \ (mod\ m)$ implies $b \equiv a \ (mod\ m)$ (symmetry).

P3. $a \equiv b \ (mod\ m)$ and $b \equiv c \ (mod\ m)$ imply $a \equiv c \ (mod\ m)$ (transitivity).

P4. If $a \equiv b \ (mod\ m)$, $c \equiv d \ (mod\ m)$, and x, y are integers, then $ax + cy \equiv bx + dy \ (mod\ m)$ (integer linear combination).

P5. If $c > 0$ then $a \equiv b \ (mod\ m)$ if, and only if, $ac \equiv bc \ (mod\ mc)$.

P6. If $a = b$, then $a \equiv b \ (mod\ m)$ for any m.

Example 27 Observe that $x \equiv 0 \ (mod\ 1)$ for any integer x. Also observe from P5 (with $c = 2$) that $\frac{1}{2}x \equiv 0 \ (mod\ 1)$ if and only if $x \equiv 0 \ (mod\ 2)$.

A *linear diophantine equation (LDE)* is a linear equation $c_1x_1 + \ldots + c_nx_n = c_0$, where $x_1, \ldots, x_n$ are integer variables and $c_0, \ldots, c_n$ are rational numbers. A variable x_i is said to *occur* in the LDE if $c_i \neq 0$. We denote a system of m LDEs in a matrix form as $CX = D$, where C denotes an $m \times n$ matrix of rationals, X denotes a column vector of n integer variables and D denotes a column vector of m rationals. When we write a (single) LDE in the form $CX = D$, it is implicitly assumed that the sizes of C, X, D are of the form $1 \times n, n \times 1, 1 \times 1$, respectively. A variable is said to *occur* in a system of LDEs if it occurs in at least one of the LDEs in the given system of LDEs.

A *linear modular equation (LME)* has the form $c_1x_1 + \ldots + c_nx_n \equiv c_0 \ (mod\ l)$, where $x_1, \ldots, x_n$ are integer variables, $c_0, \ldots, c_n$ are rational numbers, and l is a rational number. We call l the modulus of the LME. Allowing l to be a rational number allows for simpler proofs and covers the case when l is an integer. For brevity, we write a LME $t \equiv c \ (mod\ l)$ by $t \equiv_l c$. A variable x_i is said to *occur* in an LME if l does not divide c_i.

A *system* of LDEs (LMEs) denotes conjunctions of LDEs(LMEs). If F, G are a system of LDEs (LMEs), then $F \wedge G$ is also a system of LDEs (LMEs).

6.3.1 Craig Interpolants

Given two logical formulas F and G in a theory $\mathcal{T}$ such that $F \wedge G$ is unsatisfiable in $\mathcal{T}$, an *interpolant* I for the ordered pair (F, G) is a formula such that

(1) $F \Rightarrow I$ in $\mathcal{T}$

(2) $I \wedge G$ is unsatisfiable in $\mathcal{T}$

(3) I refers to only the common variables of A and B.

The interpolant I can contain symbols that are interpreted by $\mathcal{T}$. In this chapter such symbols will be one of the following: addition ($+$), equality ($=$), modular equality for some rational number m ($\equiv_m$), disequality ($\neq$), and multiplication by a rational number ($\times$). The exact set of interpreted symbols in the interpolant depends on $\mathcal{T}$.

6.4 System of Linear Diophantine Equations (LDEs)

In this section we discuss proofs of unsatisfiability and interpolation algorithm for LDEs. The following theorem from [129] gives a necessary and sufficient condition for a system of LDEs to have an integral solution.

Theorem 11 *(Corollary 4.1(a) in Schrijver [129]) A system of LDEs $CX = D$ has no integral solution for X, if and only if there exists a rational row vector R such that RC is integral and RD is not an integer.*

Definition 15 *We say a system of LDEs $CX = D$ is* **unsatisfiable** *if it has no integral solution for X. For a system of LDEs $CX = D$ a* **proof of unsatisfiability** *is a rational row vector R such that RC is integral and RD is not an integer.*

In section 6.6 we describe how a proof of unsatisfiability R can be obtained in polynomial time for an unsatisfiable system of LDEs. (We show in the appendix D.9 that R can be converted to a polynomially sized proof in a *cutting-plane* proof system [129, 44].)

Example 28 Consider the system of LDEs $CX = D$ and a proof of unsatisfiability R:

$$CX = D := \begin{bmatrix} 1 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & 2 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix} \qquad \begin{aligned} R &= [\tfrac{1}{2}, -\tfrac{1}{2}, \tfrac{1}{2}] \\ RC &= [0, 2, 1] \\ RD &= \tfrac{3}{2} \end{aligned}$$

Example 29 Consider the system of LDEs $CX = D$ and a proof of unsatisfiability R:

$$CX = D := \begin{bmatrix} 1 & -2 & 0 \\ 1 & 0 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad \begin{aligned} R &= [\tfrac{1}{2}, \tfrac{1}{2}] \\ RC &= [1, -1, -1] \\ RD &= \tfrac{1}{2} \end{aligned}$$

The above examples will be used as running examples in the chapter.

Definition 16 *(***Implication***) A system of LDEs $CX = D$* **implies** *a (single) LDE $AX = B$, if every integral vector X satisfying $CX = D$ also satisfies $AX = B$.*

Similarly, $CX = D$ **implies** *a (single) LME $AX \equiv_m B$, if every integral vector X satisfying $CX = D$ also satisfies $AX \equiv_m B$.*

Lemma 1 *(Linear combination) For every rational row vector U the system of LDEs $CX = D$ implies the LDE $UCX = UD$. Note that $UCX = UD$ is simply a linear combination of the equations in $CX = D$. The system $CX = D$ also implies the LME $UCX \equiv_m UD$ for any rational number m.*

Example 30 The system of LDEs $CX = D$ in Example 29 implies the LDE $[\frac{1}{2}, \frac{1}{2}]CX = [\frac{1}{2}, \frac{1}{2}]D$, which simplifies to $x - y - z = \frac{1}{2}$. The system $CX = D$ also implies the LME $x - y - z \equiv_m \frac{1}{2}$ for any rational number m.

6.4.1 Computing Interpolants for Systems of LDEs

Let $F \wedge G$ denote an unsatisfiable system of LDEs. The following example shows that an unsatisfiable system of LDEs does not always have an LDE as an interpolant.

Example 31 Let $F := x - 2y = 0$ and $G := x - 2z = 1$. Intuitively, F expresses the constraint that x is even and G expresses the constraint that x is odd, thus, $F \wedge G$ is unsatisfiable. We gave a proof of unsatisfiability of $F \wedge G$ in Example 29. Observe that the pair (F, G) does not have any quantifier-free interpolant that is also a LDE. The problem is that the interpolant can only refer to the variable x. We can prove (using Lemma 6 or see Appendix D.1) that there is no formula I of the form $c_1x + c_2 = 0$, where c_1, c_2 are rational numbers, such that $F \Rightarrow I$ and $I \wedge G$ is unsatisfiable.

As shown by the above example it is possible that there exists no LDE that is an interpolant for (F, G). We show that in this case the system (F, G) always has an LME as an interpolant. In the above example an interpolant will be $x \equiv_2 0$. Intuitively, the interpolant means that x is an even integer.

We now describe the algorithm for obtaining interpolants. Let $AX = A', BX = B'$ be systems of LDEs, where $X = [x_1, \ldots, x_n]$ is a column vector of n integer variables. Suppose the combined system of LDEs $AX = A' \wedge BX = B'$ is unsatisfiable. We want to compute an interpolant for $(AX = A', BX = B')$. Let $R = [R_1, R_2]$ be a proof of unsatisfiability of $AX = A' \wedge BX = B'$ according to definition 15. Then

$$R_1A + R_2B \quad \text{is integral and} \quad R_1A' + R_2B' \quad \text{is not an integer.}$$

Recall that a variable is said to *occur* in a system of LDEs if it occurs with a non-zero coefficient in one of the equations in the system of LDEs. Let $V_{AB} \subseteq X$ denote the set of variables that occur in both $AX = A'$ and $BX = B'$, let $V_{A \setminus B} \subseteq X$ denote the set of variables occurring only in $AX = A'$ (and not in $BX = B'$), and let $V_{B \setminus A} \subseteq X$ denote the set of variables occurring only in $BX = B'$ (and not in $AX = A'$).

We call the LDE $R_1AX = R_1A'$ a **partial interpolant** for $(AX = A', BX = B')$. It is a linear combination of equations in $AX = A'$. The partial interpolant $R_1AX = R_1A'$ can be written in the following form

$$\sum_{x_i \in V_{A \setminus B}} a_i x_i + \sum_{x_i \in V_{AB}} b_i x_i = c \tag{6.1}$$

where all coefficients a_i, b_i and $c = R_1A'$ are rational numbers. Observe that the partial interpolant does not contain any variable that occurs only in $BX = B'$ $(V_{B \setminus A})$.

Lemma 2 *The coefficient a_i of each $x_i \in V_{A \setminus B}$ in the partial interpolant $R_1AX = R_1A'$ (Equation 6.1) is an integer.*

Lemma 3 *The partial interpolant $R_1AX = R_1A'$ satisfies the first two conditions in the definition of an interpolant. That is,*

1. $AX = A'$ implies $R_1AX = R_1A'$

2. $(R_1AX = R_1A') \wedge BX = B'$ is unsatisfiable

If $a_i = 0$ for all $x_i \in V_{A\setminus B}$ (equation 6.1), then the partial interpolant only contains the variables from V_{AB}. In this case the partial interpolant is an interpolant for $(AX = A', BX = B')$.

The proofs of above lemmas are given in the appendix D.1.

Example 32 Consider the system of LDEs $CX = D$ in Example 28. A proof of unsatisfiability for this system is $R = [\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}]$. Let $AX = A'$ be the first two equations in $CX = D$, that is, $x + y = 1 \wedge x - y = 1$ (in matrix form). Let $BX = B'$ be the third equation in $CX = D$, that is, $2y + 2z = 3$. Observe that $V_{A\setminus B} := \{x\}, V_{AB} := \{y\}, V_{B\setminus A} := \{z\}$. In this case $R_1 = [\frac{1}{2}, -\frac{1}{2}]$. The partial interpolant for the pair $(AX = A', BX = B')$ is $y = 0$, which is also an interpolant because $y \in V_{AB}$.

The following example shows that a partial interpolant need not be an interpolant.

Example 33 Consider the system $CX = D$ in Example 29. A proof of unsatisfiability for this system is $R = [\frac{1}{2}, \frac{1}{2}]$. Let $AX = A'$ be the first equation in $CX = D$, that is, $x - 2y = 0$. Let $BX = B'$ be the second equation in $CX = D$, that is, $x - 2z = 1$. Observe that $V_{A\setminus B} := \{y\}, V_{AB} := \{x\}, V_{B\setminus A} := \{z\}$. In this case $R_1 = [\frac{1}{2}]$. Thus, the partial interpolant for the pair $(AX = A', BX = B')$ is $\frac{1}{2}x - y = 0$. Observe that the partial interpolant is not an interpolant as it contains the variable y, which does not occur in V_{AB}. This is not surprising since we have already seen in Example 31 that $(x - 2y = 0, x - 2z = 1)$ cannot have an interpolant that is a LDE.

We now intuitively describe how to remove variables from the partial interpolant that are not common to $AX = A'$ and $BX = B'$. In example 33 the partial interpolant is $\frac{1}{2}x - y = 0$, where $y \notin V_{AB}$. We show how to eliminate y from $\frac{1}{2}x - y = 0$ in order to obtain an interpolant. We use modular arithmetic in order to eliminate y. Informally, the equation $\frac{1}{2}x - y = 0$ implies $\frac{1}{2}x - y \equiv 0 \ (mod\ \gamma)$ for any rational number γ. Let α denote the greatest common divisor of the coefficients of variables (in $\frac{1}{2}x - y = 0$) that do not occur in V_{AB}. In this example $\alpha = 1$ (gcd of the coefficient of y). We know $\frac{1}{2}x - y = 0$ implies $\frac{1}{2}x - y \equiv 0 \ (mod\ 1)$. Since y is an integer variable $y \equiv 0 \ (mod\ 1)$. We can add $\frac{1}{2}x - y \equiv 0 \ (mod\ 1)$ and $y \equiv 0 \ (mod\ 1)$ to obtain $\frac{1}{2}x \equiv 0 \ (mod\ 1)$ (note that y is eliminated). Intuitively, the linear modular equation $\frac{1}{2}x \equiv 0 \ (mod\ 1)$ is an interpolant for $(x - 2y = 0, x - 2z = 1)$. By using basic modular arithmetic this interpolant can be written as $x \equiv 0 \ (mod\ 2)$.

We now formalize the above intuition to address the case when the partial interpolant contains variables that are not common to $AX = A'$ and $BX = B'$.

Theorem 12 *Assume that the coefficient a_i of at least one $x_i \in V_{A\setminus B}$ in the partial interpolant (Equation 6.1) is not zero. Let α denote the gcd of $\{a_i | x_i \in V_{A\setminus B}\}$.*

(a) α is an integer and $\alpha > 0$.

(b) Let β be any integer that divides α. Then the following linear modular equation

I_β is an interpolant for $(AX = A', BX = B')$.

$$I_\beta := \sum_{x_i \in V_{AB}} b_i x_i \equiv c \; (mod \; \beta)$$

Observe that I_β contains only variables that are common to both $AX = A'$ and $BX = B'$. It is obtained from the partial interpolant by dropping all variables occurring only in $AX = A'$ ($V_{A\setminus B}$) and replacing the linear equality by a modular equality.

The proof can be found in the appendix D.1.2. In theorem 12, I_1 is always an interpolant for $(AX = A', BX = B')$. For $\alpha > 1$ theorem 12 allows us to obtain multiple interpolants by choosing different β. For any β that divides α, $I_\alpha \Rightarrow I_\beta$ and $I_\beta \Rightarrow I_1$. Depending upon the application one can use the strongest interpolant I_α (least satisfying assignments) or the weakest interpolant I_1 (most satisfying assignments). The next example illustrates the use of Theorem 12 in obtaining multiple interpolants.

Example 34 Consider the system of LDEs $CX = D$ and a proof of unsatisfiability R:

$$CX = D := \begin{bmatrix} 30 & 4 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix} \qquad \begin{aligned} R &= [\tfrac{1}{5}, \tfrac{1}{5}] \\ RC &= [6, 1] \\ RD &= \tfrac{4}{5} \end{aligned}$$

Let $AX = A'$ be the first equation in $CX = D$, that is, $30x + 4y = 2$ (in matrix form). Let $BX = B'$ be the second equation in $CX = D$, that is, $y = 2$. Observe that $V_{A\setminus B} := \{x\}, V_{AB} := \{y\}, V_{B\setminus A} := \emptyset$. In this case $R_1 = [\frac{1}{5}]$. The partial interpolant $R_1 AX =$

R_1A' for the pair $(AX = A', BX = B')$ is $6x + \frac{4}{5}y = \frac{2}{5}$. The partial interpolant is not an interpolant as it contains the variable x, which does not occur in V_{AB}.

Using Theorem 12 we can obtain four interpolants for the pair $(AX = A', BX = B')$:

$$\begin{aligned}
I_1 &:= \frac{4}{5}y \equiv_1 \frac{2}{5} \\
I_2 &:= \frac{4}{5}y \equiv_2 \frac{2}{5} \\
I_3 &:= \frac{4}{5}y \equiv_3 \frac{2}{5} \\
I_6 &:= \frac{4}{5}y \equiv_6 \frac{2}{5}
\end{aligned}$$

I_6 implies all other interpolants. That is, $I_6 \Rightarrow I_3, I_6 \Rightarrow I_2, I_6 \Rightarrow I_1$. I_1 is implied by all other interpolants. That is, $I_2 \Rightarrow I_1, I_3 \Rightarrow I_1, I_6 \Rightarrow I_1$.

Lemma 3 and Theorem 12 give us a sound and complete algorithm for computing an interpolant for unsatisfiable systems of LDEs. The pseudocode is given in Algorithm 6.1.

The interpolant produced by Algorithm 6.1 depends on the proof of unsatisfiability. There is no guarantee that the generated interpolant will be a LDE, even if there exists an interpolant for $(AX = A', BX = B')$ that is a LDE.

6.5 System of Linear Modular Equations (LMEs)

In this section we discuss proofs of unsatisfiability and interpolation algorithm for LMEs. We first consider a system of LMEs where all equations have the same

Algorithm 6.1 Interpolation for Linear Diophantine Equations

```
Input: Systems of LDEs $AX = A'$ and $BX = B'$, $AX = A' \wedge BX = B'$ is unsatisfiable.
Output: Return an interpolant for $(AX = A', BX = B')$
1: $[R_1, R_2] \Leftarrow$ proof of unsatisfiability of $AX = A' \wedge BX = B'$
   $\{R_1A + R_2B$ is integral and $R_1A' + R_2B'$ is not an integer$\}$
2: $PI \Leftarrow R_1AX = R_1A'$ $\{PI$ represents partial interpolant$\}$
3: $PI$ can be written as
       $$\sum_{x_i \in V_{A \setminus B}} a_i x_i + \sum_{x_i \in V_{AB}} b_i x_i = c$$
   $\{V_{AB} \subseteq X$ represents variables occurring in both $AX = A', BX = B'$, while
   $V_{A \setminus B} \subseteq X$ represents variables occurring in only $AX = A'\}$
4: if $a_i = 0$ for all $x_i \in V_{A \setminus B}$ then
5:     return $PI$ $\{$Interpolant is a LDE$\}$
6: else
7:     $\alpha \Leftarrow gcd\{a_i | x_i \in V_{A \setminus B}\}$ $\{\alpha$ is an integer$\}$
8:     Let $\beta$ be any integer that divides $\alpha$. Let linear modular equation
           $$I_\beta := \sum_{i \in V_{AB}} b_i x_i \equiv_\beta c$$
9:     return $I_\beta$ $\{$Interpolant is a LME$\}$
10: end if
```

modulus l, where l is a rational number. We denote this system as $CX \equiv_l D$, where C denotes an $m \times n$ rational matrix, X denotes a column vector of n integer variables and D denotes a column vector of m rational numbers. The next theorem gives a necessary and sufficient condition for $CX \equiv_l D$ to have an integral solution.

Theorem 13 *The system $CX \equiv_l D$ has no integral solution X if and only if there exists a rational row vector R such that RC is integral, lR is integral, and RD is not an integer. Note that lR denotes the row vector obtained by multiplying each element of R by rational number l. (The size of R is $1 \times m$.)*

The proof uses reduction to LDEs. See the appendix D.2.1 for the proof.

Definition 17 *We say a system of LMEs $CX \equiv_l D$ is* **unsatisfiable** *if it has no integral solution X. A* **proof of unsatisfiability** *for a system of LMEs $CX \equiv_l D$ is a rational row vector R such that RC is integral, lR is integral, and RD is not an integer.*

Example 35 Consider the system of LMEs $CX \equiv_8 D$ and a proof of unsatisfiability R:

$$CX \equiv_8 D := \begin{bmatrix} 2 & 2 \\ 2 & 1 \\ 4 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \equiv_8 \begin{bmatrix} 4 \\ 4 \\ 4 \end{bmatrix} \qquad \begin{aligned} R &= [\tfrac{1}{4}, -\tfrac{1}{2}, -\tfrac{1}{8}] \\ RC &= [-1, 0] \\ lR &= [2, -4, -1] \\ RD &= -\tfrac{3}{2} \end{aligned}$$

Intuitively, $CX \equiv_8 D$ is unsatisfiable because we can take an integer linear combination of the given equations using lR to get a contradiction $0 \equiv_8 -12$.

Definition 18 (**Implication**) *A system of LMEs $CX \equiv_l D$* **implies** *a LME $AX \equiv_l B$, if every integral vector X satisfying $CX \equiv_l D$ also satisfies $AX \equiv_l B$.*

Lemma 4 *For every* **integral** *row vector U the system of LMEs $CX \equiv_l D$ imply $UCX \equiv_l UD$.*

6.5.1 Computing Interpolants for Systems of LMEs

Let $AX \equiv_l A'$ and $BX \equiv_l B'$ be two systems of LMEs such that $AX \equiv_l A' \wedge BX \equiv_l B'$ is unsatisfiable. We show that $(AX \equiv_l A', BX \equiv_l B')$ always has an LME as an interpolant. Let $R = [R_1, R_2]$ denote a proof of unsatisfiability for the system $AX \equiv_l A' \wedge BX \equiv_l B'$ such that $R_1A + R_2B$ is integral, $lR = [lR_1, lR_2]$ is integral, and $R_1A' + R_2B'$ is not an integer. The following theorem shows that we can take integer linear combinations of equations in $AX \equiv_l A'$ to obtain interpolants.

Theorem 14 *We assume $l \neq 0$. Let S_1 denote the set of non-zero coefficients of $x_i \in V_{A \setminus B}$ in R_1AX. Let S_2 denote the set of non-zero elements of row vector lR_1. If $S_2 = \emptyset$, then the interpolant for $(AX \equiv_l A', BX \equiv_l B')$ is a trivial LME $0 \equiv_l 0$. Otherwise, let $S_2 \neq \emptyset$. Let α denote the gcd of numbers in $S_1 \cup S_2$. (a) α is an integer and $\alpha > 0$.*
(b) Let β be any integer that divides α. Let $U = \frac{l}{\beta}R_1$. Then $UAX \equiv_l UA'$ is an interpolant for $(AX \equiv_l A', BX \equiv_l B')$.

The proof is given in the appendix D.2.2.

Example 36 Consider the system of LMEs $CX \equiv_l D$ in Example 35. Let $AX \equiv_l A'$ denote the first two equations in $CX \equiv_l D$ and $BX \equiv_l B'$ denote the last equation in $CX \equiv_l D$. Observe that $V_{A \setminus B} := \{y\}, V_{AB} := \{x\}, V_{B \setminus A} := \emptyset$. A proof of unsatisfiability for $CX \equiv_l D$ is $R = [\frac{1}{4}, -\frac{1}{2}, -\frac{1}{8}]$. We have $R_1 = [\frac{1}{4}, -\frac{1}{2}]$, $lR_1 = [2, -4]$, R_1AX is $-\frac{1}{2}x$, $S_1 = \emptyset$, $S_2 = \{2, -4\}$, $\alpha = 2$. We can take $\beta = 1$ or $\beta = 2$ to obtain two valid interpolants. For $\beta = 1$, $U = [2, -4]$ and the interpolant $UAX \equiv_l UA'$ is $-4x \equiv_8 -8$ (equivalently $x \equiv_2 0$). For $\beta = 2$, $U = [1, -2]$ and the interpolant $UAX \equiv_l UA'$ is $-2x \equiv_8 -4$ (equivalently $x \equiv_4 2$).

6.5.2 Handling LMEs with Different Moduli

Consider a system F of LMEs, where equations in F can have different moduli. In order to check the satisfiability of F, we obtain another equivalent system of equations F' such that each equation in F' has the same modulus. This is done using a standard trick described in Mathews [109]. Let $m_1, \ldots, m_k$ represent the different moduli occurring in equations in F. Let m denote the least common multiple of $m_1, \ldots, m_k$. We multiply each equation $t \equiv_{m_i} c$ in F by $\frac{m}{m_i}$ to obtain another equation $\frac{m}{m_i}t \equiv_m \frac{m}{m_i}c$. Let F' represent the set of new equations. All equations in F' have same modulus m. Using basic modular arithmetic one can show that F and F' are equivalent. Suppose F is unsatisfiable. Then the interpolants for any partition of F can be computed by working with F' and using the techniques described in the previous section. For example, let F represent the following system of LMEs $x \equiv_2 1 \wedge x + y \equiv_4 2 \wedge 2x + y \equiv_8 4$. One can work with $F' := 4x \equiv_8 4 \wedge 2x + 2y \equiv_8 4 \wedge 2x + y \equiv_8 4$ instead of F.

6.6 Algorithms for Obtaining Proofs of Unsatisfiability

Polynomial time algorithms are known for determining if a system of LDEs $CX = D$ has an integral solution or not [129]. We review one such algorithm that is based on the computation of the *Hermite normal form (HNF)* of the matrix C.

Using standard Gaussian elimination it can be determined if $CX = D$ has a rational solution or not. If $CX = D$ has no rational solution, then it cannot have any integral solution. In the discussion below we assume that $CX = D$ has a rational solution. Without loss of generality we assume that matrix C has *full row rank*, that is, all rows of C are linearly independent (linearly dependent equations can be removed).

The HNF of a $m \times n$ matrix C with full row rank is of the form $[E \;\; 0]$ where 0 represents an $m \times (n - m)$ matrix filled with zeros and E is a square $m \times m$ matrix with the following properties: 1) E is lower triangular 2) E is non-singular (invertible) 3) all entries in E are non-negative and the maximum entry in each row lies on the diagonal. The HNF of a matrix can be obtained by three elementary column operations. 1) Exchanging two columns. 2) Multiplying a column by -1. 3) Adding an integral multiple of one column to another column. Each column operation can be represented by a unimodular matrix. A *unimodular matrix* is a square matrix with integer entries and determinant +1 or -1. The product of unimodular matrices is a unimodular matrix. The inverse of a unimodular matrix is a unimodular matrix. The conversion of C to HNF can be represented as follows $CU = [E \;\; 0]$, where U is a unimodular matrix, the sizes of C, U, E are $m \times n, n \times n, m \times m$, respectively and 0 represents an $m \times (n-m)$ matrix filled with zeros ($n \geq m$ because C has full row-rank). The following result shows the use of HNF in determining the satisfiability of a system of LDEs. Let E^{-1} denotes the matrix inverse of E.

Lemma 5 *(Corollary 5.3(b) in Schrijver [129]) For C, X, D, E defined as above, $CX = D$ has no integral solution if and only if $E^{-1}D$ is not integral.*

Example 37 For the system of LDEs $CX = D$ in example 28 we have the following:

$$\underbrace{\begin{bmatrix} 1 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & 2 & 2 \end{bmatrix}}_{C} \underbrace{\begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 1 \end{bmatrix}}_{U} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}}_{E} \qquad \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ \frac{-1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} \end{bmatrix}}_{E^{-1}} \underbrace{\begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix}}_{D} = \underbrace{\begin{bmatrix} 1 \\ 0 \\ \frac{3}{2} \end{bmatrix}}_{\textit{not integral}}$$

Example 38 For the system of LDEs $CX = D$ in example 29 we have the following:

$$\underbrace{\begin{bmatrix} 1 & -2 & 0 \\ 1 & 0 & -2 \end{bmatrix}}_{C} \underbrace{\begin{bmatrix} 1 & 2 & -2 \\ 0 & 1 & -1 \\ 0 & 0 & -1 \end{bmatrix}}_{U} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 1 & 2 & 0 \end{bmatrix}}_{[E \;\; 0]} \qquad \underbrace{\begin{bmatrix} 1 & 0 \\ \frac{-1}{2} & \frac{1}{2} \end{bmatrix}}_{E^{-1}} \underbrace{\begin{bmatrix} 0 \\ 1 \end{bmatrix}}_{D} = \underbrace{\begin{bmatrix} 0 \\ \frac{1}{2} \end{bmatrix}}_{not\ integral}$$

6.6.1 Obtaining a Proof of Unsatisfiability for a System of LDEs

If a system of LDEs $CX = D$ is unsatisfiable, then we want to compute a row vector R such that RC is integral and RD is not an integer. The following corollary shows that the proof of unsatisfiability can be obtained by using the HNF of C.

Corollary 14 *Given $CX = D$ where C, D are rational matrices, and C has full row rank. Let $[E \;\; 0]$ denote the HNF of C. If $CX = D$ has no integral solution, then $E^{-1}D$ is not integral. Suppose the i^{th} entry in $E^{-1}D$ is not an integer. Let R' denote the i^{th} row in E^{-1}. Then (a) $R'D$ is not an integer and (b) $R'C$ is integral. Thus, R' serves as the required proof of unsatisfiability of $CX = D$.*

The proof is given in the appendix D.3.

Example 39 In example 37 the third row in $E^{-1}D$ is not an integer. Thus, the proof of unsatisfiability of $CX = D$ is the third row in E^{-1} which is $[0, 0, \frac{1}{2}]$.

In example 38 the second row in $E^{-1}D$ is not an integer. Thus, the proof of unsatisfiability of $CX = D$ is the second row in E^{-1} which is $[-\frac{1}{2}, \frac{1}{2}]$.

Proofs of unsatisfiability for LMEs Let $CX \equiv_l D$ be a system of LMEs. Each equation $t_i \equiv_l d_i$ in $CX \equiv_l D$ can be written as an equi-satisfiable LDE, $t_i + lv_i = d_i$, where v_i is a new integer variable. In this way we can reduce $CX \equiv_l D$ to an equi-satisfiable system of LDEs $C'Z = D$. The proof of unsatisfiability of $C'Z = D$ is exactly a proof of unsatisfiability of $CX \equiv_l D$ (see the proof of theorem 13).

Complexity If a system of LDEs or LMEs is unsatisfiable, then we can obtain a proof of unsatisfiability in polynomial time. This is because HNF computation, matrix inversion, and matrix multiplication can be done in polynomial time in the size of input [129, 133]. The interpolation algorithms described in Sections 6.4 and 6.5 are polynomial in the size of the given formulas and the proof of unsatisfiability.

6.7 Handling Linear Diophantine Equations and Disequations

We show how to compute interpolants in presence of linear diophantine disequations. A *linear diophantine disequation (LDD)* is of the form $c_1x_1 + \ldots + c_nx_n \neq c_0$, where $c_0, \ldots, c_n$ are rational numbers and $x_1, \ldots, x_n$ are integer variables. *A system of LDEs+LDDs* denotes conjunctions of LDEs and LDDs. For example, $x+2y = 1 \wedge x+y \neq 1 \wedge 2y+z \neq 1$ with x, y, z as integer variables represents a system of LDEs+LDDs. We represent a conjunction of m LDDs as $\bigwedge_{i=1}^{m} C_iX \neq D_i$, where C_i is a rational row vector and D_i is a rational number. The next theorem gives a necessary and sufficient condition for a system of LDEs+LDDs to have an integral solution.

Theorem 15 *Let F denote $AX = B \wedge \bigwedge_{i=1}^{m} C_i X \neq D_i$. The following are equivalent:*

1. F has no integral solution

2. F has no rational solution or $AX = B$ has no integral solution.

The proof of $(2) \Rightarrow (1)$ in Theorem 15 is easy. The proof of $(1) \Rightarrow (2)$ is involved and relies on the following lemmas (full proof is given in the appendix D.6). The first lemma shows that if a system of LDEs $AX = B$ has an integral solution, then every LDE that is implied by $AX = B$, can be obtained by a linear combination of equations in $AX = B$.

Lemma 6 *A system of LDEs $AX = B$ implies a LDE $EX = F$ if and only if $AX = B$ is unsatisfiable or there exists a rational vector R such that $E = RA$ and $F = RB$.*

We use the properties of the *cutting-plane* proof system [129, 44] in order to prove lemma 6. The proof is given in the appendix D.4. The next lemma shows that if a system of LDEs implies a disjunction of LDEs, then it implies one of the LDEs in the disjunction (also called *convexity* [120]).

Lemma 7 *A system of LDEs $AX = B$ implies $\bigvee_{i=1}^{m} C_i X = D_i$ if and only if there exists $1 \leq k \leq m$ such that $AX = B$ implies $C_k X = D_k$.*

We use a theorem from [129] that gives a parametric description of the integral solutions to $AX = B$ in order to prove lemma 7. See the appendix D.5 for the full proof. Let F denote $AX = B \wedge \bigwedge_{i=1}^{m} C_i X \neq D_i$. Using Theorem 15 we can determine whether F has an integral solution in polynomial time. This is because checking if $AX = B$ has an integral solution can be done in polynomial time [129, 44]. Checking whether the system F has a rational solution can be done in polynomial time as well [120].

6.7.1 Interpolants for LDEs+LDDs

We say a system of LDEs+LDDs is **unsatisfiable** if it has no integral solution. Consider systems of LDEs+LDDs $F := F_1 \wedge F_2$ and $G := G_1 \wedge G_2$, where F_1, G_1 are systems of LDEs and F_2, G_2 are systems of LDDs. $F \wedge G$ represents another system of LDEs+LDDs. Suppose $F \wedge G$ is unsatisfiable. The interpolant for (F, G) can be computed by considering two cases (due to theorem 15):

Case 1: $F \wedge G$ is unsatisfiable because $F_1 \wedge F_2 \wedge G_1 \wedge G_2$ has no rational solution. We can compute an interpolant for (F, G) using the techniques described in [113, 128, 54]. The algorithms in [113, 128, 54] can result in interpolants containing inequalities. We describe an alternative algorithm in the appendix D.7 that always produces a LDE or a LDD as an interpolant.

Case 2: $F \wedge G$ is unsatisfiable because $F_1 \wedge G_1$ has no integral solution. In this case we can compute an interpolant for the pair (F_1, G_1) using the techniques from Section 6.4. The interpolant for (F_1, G_1) will be an interpolant for (F, G). It can be a LDE or a LME.

Example	Preds/Interpolants	VINT2
ex1	$y \equiv_2 1$	2.72s
ex2	$x+y \equiv_2 0$	0.83s
ex4	$x+y+z \equiv_4 0$	0.95s
ex5	$x \equiv_4 0, y \equiv_4 0$	1.1s
ex6	$4x+2y+z \equiv_8 0$	0.93s
ex7	$4x-2y+z \equiv_{2^{22}} 0$	0.54s
forb1	$x+y \equiv_3 0$	-

Table 6.1: Table showing the predicates needed and time taken in seconds.

6.8 Experimental Results

We implemented the interpolation algorithms for conjunctions of LDEs, LMEs, LDDs in a tool called `INT2 (INTeger INTerpolate)`. The experiments are performed on a 1.86 GHz Intel Xeon (R) machine with 4 GB of memory running Linux. `INT2` is designed for computing interpolants for formulas (LDEs, LMEs, LDEs+LDDs) that are satisfiable over rationals but unsatisfiable over integers. Currently, there are no other interpolation tools for such formulas.

6.8.1 Use of Interpolants in Verification

We wrote a collection of small C programs each containing a `while` loop and an ERROR label. These programs are safe (ERROR is unreachable). The existing tools based on predicate abstraction and counterexample guided abstraction refinement (CEGAR) such as `BLAST` [2, 89], `SATABS` [16] are not able to verify these programs. This is because the inductive invariant required for the proof contains LMEs as predicates, shown in the "Preds/Interpolants" column of Table

6.1. These predicates cannot be discovered by the interpolation engine [113, 128] used in `BLAST` or by the weakest precondition based procedure used in `SATABS`. The interpolation algorithms described in this chapter are able to find the right predicates by computing the interpolants for spurious program traces. Only one unwinding of the `while` loop suffices to find the right predicates in 6 out of 7 cases. In program ex5 multiple unwindings of the `while` loop produces predicates of the form $x = 0, y = 4, x = 4, y = 8, \ldots$. After a few unwindings these predicates are generalized to obtain $x \equiv_4 0, y \equiv_4 0$[1].

We wrote similar programs in Verilog and tried verifying them with `VCEGAR` [23], a CEGAR based model checker for Verilog. `VCEGAR` fails on these examples due to its use of weakest preconditions. Next, we externally provided the interpolants (predicates) found by `INT2` to `VCEGAR`. With the help of these predicates `VCEGAR` is able to show the unreachability of ERROR labels in all examples except forb1 (ERROR is reachable in the Verilog version of forb1). The runtimes are shown in "VINT2" column.

Müller-Olm and Seidl [118] propose an abstraction technique that can infer linear invariants that are sound with respect to integer arithmetic modulo a power of 2. Their work provides an alternative way of verifying the programs listed in Table 6.1.

[1]The generalization was done manually but can be automated as follows: on seeing a sequence of predicates $t = c_1, t = c_2, \ldots$ add a predicate $t = 0\ (mod\ gcd(c_1, c_2, \ldots))$ where t is a term and $c_1, c_2, \ldots$ are constants.

6.8.2 Proofs of Unsatisfiability (PoU) Algorithms

We obtained 459 unsatisfiable formulas (system of LDEs) by unwinding the `while` loops for C programs mentioned above. The number of LDEs in these formulas range from 3 to 1500 with 2 to 4 variables per equation. There are two options for obtaining PoU in INT2.

(a) Using Hermite Normal Form (HNF) (Section 6.6.1). We use PARI/GP [136] to compute HNF of matrices.

(b) By using a state-of-the-art SMT solver Yices 1.0.11 [24] in a black-box fashion (along the lines of [128]). Given a system of LDEs $AX = B$ we encode the constraints that RA is integral and RB is not an integer by means of mixed integer linear arithmetic constraints (see the appendix D.10). The SMT solver returns concrete values to elements in R if $AX = B$ is unsatisfiable.

The comparison between (a) and (b) is shown in Figure 6.1. There is a timeout of 1000 seconds per problem. The HNF based algorithm is able to solve all problems, while the black-box usage of Yices cannot solve 102 problems within the timeout. Thus, the HNF based method is superior over the black-box use of Yices.

We also ran Yices to decide whether $AX = B$ has an integral solution or not. The system $AX = B$ (X integral) is given to Yices. In this case, Yices is very efficient and reports the satisfiability or unsatisfiability of $AX = B$ quickly. However, no PoU is provided when $AX = B$ is unsatisfiable. In principle it is possible for

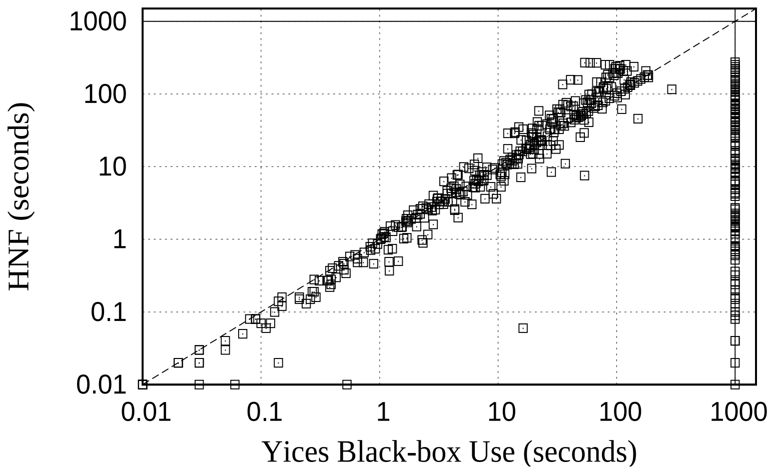


Figure 6.1: Comparing Hermite Normal Form based algorithm and black-box use of Yices for getting proofs of unsatisfiability

Yices to provide a PoU when $AX = B$ is unsatisfiable (although this will add some overhead).

Note that the interpolation algorithms proposed in this chapter are independent of the algorithm used to generate the PoU. Any decision procedure that can produce PoU according to definitions 15, 17 can be used (we are not restricted to using HNF or Yices).

6.9 Chapter Summary

We presented polynomial time algorithms for computing proofs of unsatisfiability and interpolants for conjunctions of linear diophantine equations, linear modular equations and linear diophantine disequations. These interpolation algorithms are useful for discovering modular/divisibility predicates from spurious counterexamples in a counterexample guided abstraction refinement framework.

Chapter 7

Epilogue: Future Work

The domain of hardware and software verification abounds with many challenging problems. In this dissertation I focused on some of these problems and presented possible solutions to them. It is now time to look at the possible directions for future research.

- *Proof Generation from Non-Clausal SAT Solvers:* Modern SAT solvers provide a proof of unsatisfiability for formulas that are unsatisfiable. The proofs of unsatisfiability are very useful in various verification techniques such as abstraction-refinement, proof-based abstraction, and interpolation. A promising research direction is to add proof generation capabilities to the non-clausal SAT solvers discussed in this thesis.

- *Non-clausal Learning Schemes:* Conflict driven learning is an important part of modern SAT solvers. The learning schemes used in this thesis are *clausal*, that is, the learned facts are clauses. This introduces an asymmetry in our SAT algorithms because the original formula is processed in the non-clausal form, while the learned clauses are processed in the clausal form. Despite this asymmetry the contribution of the original non-clausal formula remains significant because most of the conflicts and implications during BCP occur due to the original formula. This is partly due to the fact that a significant portion of learned clauses is discarded periodically in order to save memory and BCP time.

An interesting research direction is to come up with learning schemes that can learn more complex formulas from conflicts. Such non-clausal learned formulas can be added to the vpgraph/hpgraph directly. One way to perform non-clausal learning is to combine learned clauses that share common literals to produce new hpgraph/vpgraph components. An alternative idea is to modify the DPLL algorithm so that the branching is allowed on ϕ and $\neg\phi$, where ϕ can be a complex formula. The idea of introducing lemmas of the form $\phi \vee \neg\phi$, where ϕ can be a complex formula has been used in theorem proving based on vertical path forms [123].

- *Quantified Boolean Formulas (QBF) Solvers:* Many practical problems in verification and planning can be framed as QBF formulas. The Boolean satisfiability (SAT) problem can be regarded as a restricted form of QBF, where only existential quantifiers are allowed. Unlike SAT solvers, the QBF solvers [149] can only handle small instances. Zhang et al. [147] report that the use of both CNF and DNF representations of a given Boolean formula is crucial for obtaining efficient QBF solvers. The graphical representations hpgraph/vpgraph encode the CNF/DNF representation of NNF formulas compactly and can lead to efficient QBF solvers. Recent work by Lonsing and Biere [105] also motivates the use of NNF for QBF solving.

- *Word-Level/Satisfiability Modulo Theories (SMT) Solvers:* The formulas arising in various applications are usually a Boolean combination of constraints. These constraints can range over theories such as difference logic, linear arithmetic over reals/integers, uninterpreted functions, and so on. It is inefficient to encode such problems as bit-level (propositional) formulas. In order to check the satisfiability of these formulas, SMT (Satisfiability Modulo Theory) solvers [83, 46, 121, 141, 122, 74] are emerging as a better option. Most of the existing SMT solvers use a CNF SAT solver for handling the Boolean structure of a given formula. It will be interesting to explore the use of non-clausal SAT solvers when reasoning about the Boolean structure in a SMT solver. See [134] for recent work in this direction. More tighter integration between various theory solvers and hpgraph/vpgraph representation is also possible.

- *Bit-vector Arithmetic Solvers:* Most hardware and software verification techniques generate decision procedure queries in bit-vector arithmetic logic. The formulas in this logic contain finite precision variables (bit-vectors), arithmetic operations over bit-vectors, and bit-wise operations (such as concatenation, extraction, shifting) over bit-vectors. In bit-blasting a given bit-vector arithmetic formula is converted to an equi-satisfiable propositional logic formula. The propositional logic formula is then checked for satisfiability using a Boolean satisfiability (SAT) solver. This is the most commonly used technique for deciding bit-vector arithmetic formulas. This technique is very successful due to the significant improvements in the capacity of SAT solvers over the past decade. The main disadvantage with the bit-blasting approach is that the high-level structure present in a word-level bit-vector formula gets lost at the propositional level. Reasoning about the propositional encodings of operators such as multiplication/division is difficult for propositional SAT solvers. As the datapath (register width) increases the corresponding SAT problems become harder. Recent work [49, 107, 82, 48] addresses these limitations by eliminating or reducing the need for bit-blasting. More research needs to be done in order to handle non-linear operations such as multiplication/division efficiently. Another promising direction is to use the non-clausal SAT solvers in a bit-blasting approach for deciding bit-vector arithmetic formulas.

- *Interpolating Theorem Provers:* Modern hardware and software verification techniques expect the decision procedures to provide proofs of unsatisfiability and interpolants. Generating interpolants for integer linear arithmetic and bit-vector arithmetic is a challenging task. In this thesis we presented efficient interpolation algorithms for subsets of integer linear arithmetic. One direction for future research is to use branch-and-cut algorithms for generating proofs of unsatisfiability and interpolants for full integer linear arithmetic. In principle one can also reduce many bit-vector arithmetic formulas to integer linear arithmetic formulas [45]. Thus, an interpolating theorem prover for integer linear arithmetic can also be used to obtain interpolants for bit-vector arithmetic formulas.

- *Combination of Abstraction Techniques:* In most predicate abstraction and CEGAR based tools, spurious behavior in the abstract model is removed by adding new predicates or making the relationships between existing predicates more precise. Thus, even the information that can be discovered efficiently using other abstract domains is learned only through multiple refinement iterations in form of new predicate relationships. Large number of predicates pose problem as both the predicate abstraction computation and the model checking of abstraction is exponential in the number of predicates. This motivates the need for combining various abstraction techniques [78, 93, 140].

  It context of circuits it maybe beneficial to combine predicate abstraction with memory abstraction techniques [80, 107] and symbolic trajectory evaluation [25] in order to handle large memories efficiently. The abstraction techniques based on uninterpreted functions can help in dealing with large datapaths [28, 27]. Finally, some combination of bit-level abstraction techniques [102, 142, 53, 115, 86, 113, 87] and word-level abstraction techniques [50, 28, 95, 140, 43] is needed in order to handle industrial circuits.

Bibliography

[1] AIGER, `http://fmv.jku.at/aiger`. 4.10, 4.10.1

[2] BLAST 2.4 website. `http://mtc.epfl.ch/software-tools/blast/`. 1.2.4, 6.8.1

[3] Cadence smv. `http://www.cadence.com/webforms/cbl_software/index.aspx`. 5.4.2, 5.6

[4] CMUSAT sat solver description, `http://www.cs.cmu.edu/~hjain/papers/cmusat-solvers.pdf`. 4.10.2

[5] EBMC website, `http://www.verify.ethz.ch/ebmc/`. 1

[6] Edimacs format. `www.satcompetition.org/2005/edimacs.pdf`. 3.8

[7] Hardware model checking competition, `http://fmv.jku.at/hwmcc07/`. 4.10.1

[8] Minisat sat solver. `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/`. 1.1, 1.2.1, 3.8, 4.10.2, 5.6

[9] MiniSAT++ sat solver description, `http://www-sr.informatik.uni-tuebingen.de/sat-race-2008/descriptions/solver_26.pdf`. 4.10.2, 4.10.3

[10] M.N. Velev, `http://www.ece.cmu.edu/~mvelev`. 4.10.1

[11] Nusmv model checker. `http://nusmv.irst.itc.it/`. 5.4.2

[12] Opencores. `http://www.opencores.org/`. 5.6.1

[13] Picosat sat solver. `http://fmv.jku.at/picosat/`. 1.1, 4.10.2

[14] Rsat sat solver. `http://reasoning.cs.ucla.edu/rsat/`. 1.1, 4.10.2

[15] SAT competition 2007, `http://www.satcompetition.org/2007/`. 4.10.1

[16] SATABS 1.9 website, `http://www.verify.ethz.ch/satabs/`. 6.8.1

[17] SatMate website. `http://www.cs.cmu.edu/~modelcheck/satmate`. 3.8

[18] Siege sat solver. `http://www.cs.sfu.ca/~loryan/personal`. 1.1, 3.8

[19] SMV2QBF, `http://fmv.jku.at/smv2qbf`. 4.10.1

[20] Sun picojava. `http://www.sun.com/processors/technologies.html`. 5.6.1

[21] TPS and ETPS. `http://gtps.math.cmu.edu/tps-papers.html`. 3

[22] UCLID verification tool, `http://www.cs.cmu.edu/~uclid/`. 3.8, 4.10.1

[23] VCEGAR 1.3 website. `http://www.cs.cmu.edu/~modelcheck/vcegar/`. 1.4, 5.6, 6.8.1

[24] Yices 1.0.11 website. `http://yices.csl.sri.com/`. 6.8.2, D.10

[25] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *Design automation conference*, pages 538–541, 1998. 7

[26] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic Reachability Analysis Based on SAT-Solvers. In *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 411–425, London, UK, 2000. Springer-Verlag. 1.2.1

[27] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Asia South Pacific design automation conference*, pages 19–24, 2006. 1.2.4, 7

[28] Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of Verilog models. In *Proceedings of the* 41st *Annual Conference on Design Automation (DAC)*, pages 218–223. ACM Press, 2004. 1.2.4, 7

[29] Peter B. Andrews. Theorem Proving via General Matings. *J. ACM*, 28(2):193–214, 1981. 1.1.1, 2, 3

[30] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof.* Kluwer Academic Publishers, Dordrecht, second edition, 2002. 1.1.1, 2, 2.2, 1, 2.2, 2.2

[31] Domagoj Babić and Alan J. Hu. Calysto: Scalable and Precise Extended Static Checking. In *30th International Conference on Software Engineering, ICSE 2008, May 10–18, 2008, Proceedings*, 2008. 1.1

[32] Domagoj Babić and Madanlal Musuvathi. Modular Arithmetic Decision Procedure. Technical Report TR-2005-114, Microsoft Research Redmond, 2005. 1.2.4, 6.2

[33] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *The* 8th *International SPIN Workshop on Model Checking of Software*, pages 103–122, 2001. 1.2.1, 1.2.3, 5, 5.1

[34] T. Ball and S.K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research, February 2000. 5

[35] Thomas Ball, Byron Cook, Satyaki Das, and Sriram Rajamani. Refining approximations in software predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 388–403. Springer, 2004. 5

[36] Thomas Ball, Byron Cook, Shuvendu K. Lahiri, and Lintao Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Computer Aided Verification*. Springer, 2004. 5

[37] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213, New York, NY, USA, 2001. ACM. 5

[38] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 522–527, New York, NY, USA, 1998. ACM Press. 1.2.4, 6.2

[39] Jason Baumgartner, Andreas Kuehlmann, and Jacob A. Abraham. Property checking via structural analysis. In *Computer Aided Verification*, pages 151–165. Springer-Verlag, 2002. 1.2.4

[40] Wolfgang Bibel. On Matrices with Connections. *J. ACM*, 28(4):633–645, 1981. 3

[41] Amin Biere. Picosat essentials. *Journal on Boolean Satisfiability, Boolean Modeling and Computation (JSAT)*, 2008. 1.1

[42] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Yhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, 1999. 1.1, 1.2.1, 5.4.2

[43] Per Bjesse. A practical approach to word level model checking of industrial netlists. In *CAV*, pages 446–458, 2008. 7

[44] Alexander Bockmayr and Volker Weispfenning. Solving numerical constraints. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 751–842. 2001. 6.1, 6.4, 6.7, 6.7, D.4

[45] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Ziyad Hanna, Zurab Khasidashvili, Amit Palti, and Roberto Sebastiani. Encoding RTL Constructs for MathSAT: a Preliminary Report. *Electr. Notes Theor. Comput. Sci.*, 144(2):3–14, 2006. 7

[46] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi A. Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *TACAS*, pages 317–333, 2005. 7

[47] Robert Brummayer and Armin Biere. C32sat: Checking c expressions. In *CAV*, pages 294–297, 2007. 4.10.1

[48] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzen, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A Lazy and Layered SMT(BV) Solver for Hard Industrial Verification Problems. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag. 1.2.4, 6.2, 7

[49] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007. 1.2.4, 6.2, 7

[50] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Computer Aided Verification,* 14^{th} *International Conference, CAV 2002*, pages 78–92, 2002. 1.2.4, 7

[51] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986. 1.2.1

[52] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992. 1.2.1

[53] Pankaj Chauhan, Edmund M. Clarke, James H. Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model

checking large state spaces using SAT based conflict analysis. In *Formal Methods in Computer Aided Design*, pages 33–51, 2002. 1.1, 7

[54] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Efficient interpolation in satisfiability modulo theories. In *TACAS*, 2008. To appear. 1.3, 1.3.2, 6.1, 6.2, 6.7.1, D.7

[55] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and Veith H. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169. Springer, 2000. 1.2.1, 5.4

[56] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5), 2003. 6.1

[57] E. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Principles of Programming Languages*, 1992. 5.3

[58] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999. 1.2.1, 1.2.4, 5.3, 5.4.1

[59] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods In System Design*, 25, 2004. 5, 5.1, 5.3

[60] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, volume 131 of *LNCS*. Springer, 1981. 1.2.1

[61] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003. 1.2.1

[62] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005. 1.1, 5

[63] Edmund Clarke, Muralidhar Talupur, and Dong Wang. SAT based predicate abstraction for hardware verification. In *In Theory and Applications of Satisfiability Testing (SAT)*, 2003. 1.2.4, 5.1, 5.5, 5.5.2

[64] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *TACAS*, pages 168–176, 2004. 1.1

[65] Edmund M. Clarke and Helmut Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice*, pages 208–224, 2003. 5.3

[66] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158, 1971. 1.1

[67] William Craig. Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. *J. Symb. Log.*, 22(3):250–268, 1957. 1.3

[68] David Cyrluk, M. Oliver Möller, and Harald Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 60–71, London, UK, 1997. Springer-Verlag. 6.2

[69] Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Proceedings of LICS*, 2001. 23

[70] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962. 1.1, 2, 4

[71] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960. 1.1, 2, 4

[72] Flavio M. de Paula and Alan J. Hu. An effective guidance strategy for abstraction-guided simulation. In *DAC*, June 4–8 2007. 5.6.1

[73] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003. 5

[74] Bruno Dutertre and Leonardo Mendonça de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, pages 81–94, 2006. 6.2, 7

[75] Niklas Eén and Armin Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *SAT*, pages 61–75, 2005. 1.1, 2.1

[76] Niklas Eén, Alan Mishchenko, and Niklas Sörensson. Applying Logic Synthesis for Speeding Up SAT. In *SAT*, pages 272–286, 2007. 4.10.3

[77] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *SAT*, pages 502–518, 2003. 1.1

[78] Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. Joining dataflow with predicates. In *ESEC/SIGSOFT FSE*, pages 227–236, 2005. 7

[79] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining Strengths of Circuit-based and CNF-based Algorithms for a High-performance SAT solver. In *DAC*, 2002. 1.1

[80] Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient modeling of embedded memories in bounded model checking. In *Computer Aided Verification*, pages 440–452, 2004. 7

[81] Vijay Ganesh, Sergey Berezin, and David L. Dill. A decision procedure for fixed-width bit-vectors. Technical Report CSTR 2007-06, Stanford Computer Science Department, 2005. 6.2

[82] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag. 1.2.4, 6.2, 7

[83] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In R. Alur and D. Peled, editors, *16th*

International Conference on Computer Aided Verification, CAV'04, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004. 7

[84] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust Sat-Solver. In *DATE*, 2002. 1.1, 3.8

[85] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV)*, volume 1254, pages 72–83, 1997. 1.2.3, 5, 5.3

[86] Aarti Gupta, Malay Ganai, Zijiang Yang, and Pranav Ashar. Iterative abstraction using SAT-based BMC with proof analysis. In *International conference on Computer-aided design (ICCAD)*, page 416, 2003. 1.1, 1.2.2, 7

[87] Anubhav Gupta. *Learning Abstractions for Model Checking*. PhD thesis, Carnegie Mellon University, 2006. 1.1, 1.2.2, 7

[88] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002. 5.1, 5.4.2

[89] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–244. ACM Press, 2004. 1.2.4, 1.3, 6, 6.8.1

[90] F. Ivančić, I. Shlyakhter, A. Gupta, Malay K. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking C programs using F-SOFT. In *International Conference on Computer Design (ICCD 2005)*. IEEE, 2005. 1.1, 5

[91] Himanshu Jain, Constantinos Bartzis, and Edmund M. Clarke. Satisfiability checking of non-clausal formulas using general matings. In *9th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 75–89, 2006. 1.4

[92] Himanshu Jain, Edmund M. Clarke, and Orna Grumberg. Efficient Craig Interpolation for Linear Diophantine (Dis)Equations and Linear Modular Equations. In *20th International Conference on Computer Aided Verification (CAV)*, 2008. 1.4

[93] Himanshu Jain, Franjo Ivancic, Aarti Gupta, Ilya Shlyakhter, and Chao Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *CAV*, pages 137–151, 2006. 7

[94] Himanshu Jain, Franjo Ivančić, Aarti Gupta, and Malay K. Ganai. Localization and register sharing for predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 397–412, 2005. 5.5.2

[95] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund Clarke. Word Level Predicate Abstraction and Refinement for Verifying RTL Verilog. In *Design Automation Conference (DAC)*, June 2005. 1.4, 7

[96] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund Clarke. VCEGAR: Verilog counterexample guided abstraction refinement. In *Proceedings of TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 583–586. Springer, 2007. 1.4

[97] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund Clarke. Word level predicate abstraction and refinement for verifying RTL Verilog. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(2):366–379, Feb. 2008. 1.4

[98] Matti Järvisalo, Tommi Junttila, and Ilkka Niemelä. Unrestricted vs restricted cut in a tableau method for boolean circuits. *Annals of Mathematics and Artificial Intelligence*, 44(4):373–399, 2005. 1.1

[99] Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006. 1.3, 6

[100] Deepak Kapur, Rupak Majumdar, and Calogero G. Zarba. Interpolation for data structures. In *SIGSOFT '06/FSE-14*, pages 105–116. ACM, 2006. 1.3, 6.2

[101] Daniel Kroening and Georg Weissenbacher. Lifting propositional interpolants to the word-level. In *FMCAD*, pages 85–89. IEEE, 2007. 1.3, 6.2

[102] R.P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach.* Princeton University Press, 1994. 1.2.1, 1.2.2, 7

[103] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *Computer-Aided Verification (CAV)*, number 2725, pages 141–153. Springer, 2003. 1.1, 5

[104] Shuvendu K. Lahiri and Randal E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV*, pages 135–147, 2004. 1.2.4, 2

[105] Florian Lonsing and Armin Biere. Nenofex: Expanding NNF for QBF Solving. In *11th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'08)*, 2008. 7

[106] Feng Lu, Li-C. Wang, Kwang-Ting Cheng, and Ric C.-Y. Huang. A Circuit SAT Solver With Signal Correlation Guided Learning. In *DATE*, pages 10892–10897, 2003. 1.1

[107] P. Manolios, S. K. Srinivasan, and D. Vroon. Automatic memory reductions for RTL-level verification. In *ICCAD*, 2006. 1.2.4, 6.2, 7

[108] Joao P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, November 1996. 1.1, 1.2.1

[109] George Ballard Mathews. *Theory of numbers*. NY, Chelsea Pub. Co., 2nd edition, 1927. 6.5.2

[110] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *AAAI*, pages 321–326, Providence, Rhode Island, 1997. 1.1

[111] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1993. 1.2.1

[112] Kenneth L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*, pages 1–13, 2003. 1.1, 1.2.2, 1.3, 1, 5.6.1, 6

[113] Kenneth L. McMillan. An interpolating theorem prover. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 16–30, 2004. 1.2.4, 1.3, 1.3.2, 6.2, 6.7.1, 6.8.1, 7, D.7

[114] Kenneth L. McMillan. Lazy Abstraction with Interpolants. In *CAV*, pages 123–136, 2006. 1.3

[115] K.L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *TACAS 2003*. Springer, 2003. 1.1, 1.2.2, 7

[116] Andreas Meier and Volker Sorge. A New Set of Algebraic Benchmark Problems for SAT Solvers. In *SAT*, pages 459–466, 2005. 3.8

[117] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC'01)*, pages 530–535, June 2001. 1.1, 1.2.1, 1.2.4, 3.7, 3.8, 4.2, 4.4.1

[118] Markus Müller-Olm and Helmut Seidl. Analysis of modular arithmetic. *ACM Trans. Program. Lang. Syst.*, 29(5):29, 2007. 6.8.1

[119] Kedar S. Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In *Computer Aided Verification*, number 1855 in LNCS, 2000. 5.1

[120] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979. 6.1, 6.2, 6.7, 6.7, D.7

[121] R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In K. Etessami and S. Rajamani, editors, *17th International Conference on Computer Aided Verification, CAV'05*, volume 3576 of *Lecture Notes in Computer Science*, pages 321–334. Springer, 2005. 7

[122] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006. 7

[123] Frank Pfenning and Dan Nesmith. Presenting intuitive deductions via symmetric simplification. In *CADE-10: Proceedings of the tenth international conference on Automated deduction*, pages 336–350, New York, NY, USA, 1990. Springer-Verlag New York, Inc. 7

[124] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3), 1986. 1.1, 2.1, 3.8, 4.10.2

[125] Mojżesz Presburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Sprawozdanie z I Kongresu metematyków slowiańskich, Warszawa 1929*, pages 92–101,395, Warsaw, Poland, 1930. Annotated English version in [132]. 6.2

[126] Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symb. Log.*, 62(3):981–998, 1997. 6.2, D.8.1

[127] A. Biere R. Brummayer. Local two-level and-inverter graph minimization without blowup. In *2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'06)*, 2006. 4.10.1

[128] Andrey Rybalchenko and Viorica Sofronie-Stokkermans. Constraint solving for interpolation. In *VMCAI*, pages 346–362, 2007. 1.3, 1.3.2, 6.2, 6.7.1, 6.8.1, 6.8.2, D.7, D.10

[129] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, NY, 1986. 6.1, 6.2, 6.3, 6.4, 11, 6.4, 6.6, 5, 6.6.1, 6.7, 6.7, D.4, 16, 17, 18, 19

[130] Mary Sheeran, Satnam Singh, and Gunnar Stalmarck. Checking safety properties using induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design*, pages 108–125, 2000. 1.1, 4.10.1, 1

[131] João P. Marques Silva. Improvements to the implementation of interpolant-based model checking. In *CHARME*, pages 367–370, 2005. 1

[132] R. Stansifer. Presburger's article on integer arithmetic: Remarks and translation. Technical Report TR84–639, Cornell University Computer Science Department, 1984. 125

[133] Arne Storjohann and George Labahn. Asymptotically fast computation of Hermite normal forms of integer matrices. In *ISSAC '96: Proceedings of the 1996 international symposium on Symbolic and algebraic computation*, pages 259–266, 1996. 6.6.1

[134] Philippe Suter. Non-Clausal Satisfiability Modulo Theories. Master's thesis, École Polytechnique Fédérale de Lausanne, 2008. 7

[135] K. Takamizawa, T. Nishizeki, and N. Saito. Linear-time computability of combinatorial problems on series-parallel graphs. *J. ACM*, 29(3):623–641, 1982. 2.3.2

[136] The PARI Group. *PARI/GP, Version 2.3.2*, 2006. `http://pari.math.u-bordeaux.fr/`. 6.8.2

[137] Christian Thiffault, Fahiem Bacchus, and Toby Walsh. Solving Non-clausal Formulas with DPLL Search. In *SAT*, 2004. 1.1

[138] G.S. Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125, 1968. 1.1, 2.1, 4.10.2

[139] VIS model checker. `http://vlsi.colorado.edu/~vis`. 5.6.2

[140] C. Wang, H. Kim, and A. Gupta. Hybrid cegar: Combining variable hiding and predicate abstraction. In *In International Conference on Computer Aided Design (ICCAD'07)*, 2007. 1.2.4, 7

[141] Chao Wang, Franjo Ivančić, Malay K. Ganai, and Aarti Gupta. Deciding Separation Logic Formulae by SAT and Incremental Negative Cycle Elimination. In *LPAR*, pages 322–336, 2005. 7

[142] D. Wang, P. Ho, J. Long, J. Kukula, Y. Zhu, T. Ma, and R. Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *DAC*, pages 35–40, 2001. 1.2.2, 7

[143] Yichen Xie and Alexander Aiken. Saturn: A SAT-Based Tool for Bug Detection. In *CAV*, pages 139–143, 2005. 1.1

[144] Greta Yorsh and Madanlal Musuvathi. A combination method for generating interpolants. In *CADE*, pages 353–368, 2005. 1.3, 6.2, D.7

[145] H. Zhang. Sato: An efficient propositional prover. In *CADE-14*, pages 272–275, 1997. 1.1

[146] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formulas. In 6th *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2003. 5.5.1

[147] Lintao Zhang. Solving QBF by Combining Conjunctive and Disjunctive Normal Forms. In *AAAI*, 2006. 7

[148] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001. 3.5, 3.6

[149] Lintao Zhang and Sharad Malik. Conflict driven learning in a quantified Boolean Satisfiability solver. In *ICCAD*, pages 442–449, 2002. 7

Appendix A

Improved Construction of Graphical Representations

In the construction of vpgraph described in chapter 2 new edges are created when we take conjunction of two formulas. In particular, when we compute vpgraph for $\phi_1 \wedge \phi_2$, every leaf in vpgraph of ϕ_1 is connected to every root in vpgraph of ϕ_2. This leads to $|L_1| \times |R_2|$ new edges, where L_1 denotes the set of leafs in ϕ_1 and R_2 denotes the set of roots in ϕ_2. Thus, the construction of vpgraph described in chapter 2 can take $O(k^2)$ time/space in the worst case where k is the size of the given formula.

Example 40 Consider a CNF formula $\phi_1 = (x_1 \vee \ldots \vee x_l) \wedge (y_1 \vee \ldots \vee y_l)$. Observe that size of ϕ_1 is linear in l. The vpgraph of ϕ_1 contains l^2 edges of the form (i, j) where $1 \leq i \leq l, 1 \leq i \leq 2 \times l$. The vpgraph is shown in Fig. A.1(a).

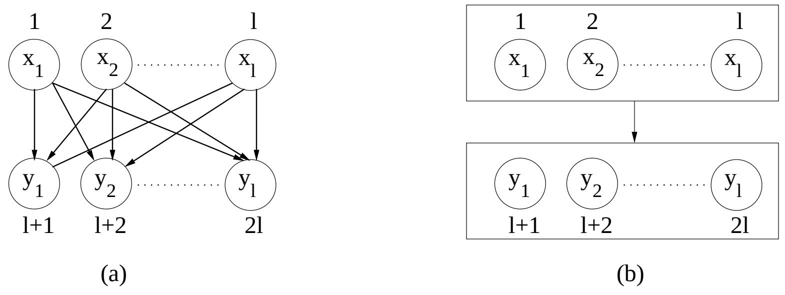


Figure A.1: Vpgraph for DNF formula $(x_1 \wedge \ldots \wedge x_l) \vee (y_1 \wedge \ldots \wedge y_l)$. (a) Explicitly representing l^2 edges. (b) Implicitly representing l^2 edges using a hyperedge from $\{1, \ldots, l\}$ to $\{l+1, \ldots, 2 \times l\}$.

In the following we present a procedure for constructing vpgraph that takes $O(k)$ time/space in the worst case where k is the size of the given formula. The basic idea is as follows: instead of explicitly adding $|L_1| \times |R_2|$ edges during a conjunction, we create one *hyperedge* (L_1, R_2). A hyperedge (L, R) *implicitly* represents that there is an edge (n, m) from every node $n \in L$ to every node $m \in R$. The representation of a hyperedge takes $O(|L| + |R|)$ space and it represents $|L| \times |R|$ edges implicitly.

Example 41 Consider a CNF formula $\phi_1 = (x_1 \vee \ldots \vee x_l) \wedge (y_1 \vee \ldots \vee y_l)$. The vpgraph ϕ_1 can be represented in $O(l)$ space by using one hyperedge for the form (A, B), where $A = \{1, \ldots, l\}, B = \{l+1, \ldots, 2 \times l\}, Lit(i) = x_i, Lit(j) = y_j, 1 \leq i \leq l, l+1 \leq j \leq 2 \times l$. Representing (A, B) requires $O(l)$ space. It implicitly represents l^2 edges which were created explicitly by the construction in chapter A. The new vpgraph is shown in Figure A.1(b).

We formalize the improved vpgraph construction below. As in the chapter 2, the vpgraph $G_v(\phi)$ of a NNF formula ϕ is defined as a tuple (V, R, L, E, Lit), where V is the set of nodes, $R \subseteq V$ is a set of root nodes, $L \subseteq V$ is a set of leaf nodes, $Lit(n)$ denotes the literal associated with node $n \in V$. The main change is that the $E \subseteq \mathcal{P}(V) \times \mathcal{P}(V)$ is the set of hyperedges.

Given ϕ, we can construct the improved vpgraph $G_v(\phi) = (V, R, L, E, Lit)$ inductively as described in the chapter 2. The only difference is in the case when we handle conjunction of two sub-formulas.

If $\phi = \phi_1 \wedge \phi_2$, then the vpgraph for ϕ is obtained by concatenating the vpgraph of ϕ_1 with the vpgraph of ϕ_2. Let $G_v(\phi_1) = (V_1, R_1, L_1, E_1, Lit_1)$ and $G_v(\phi_2) = (V_2, R_2, L_2, E_2, Lit_2)$. Then $G_v(\phi)$ contains all the nodes and edges in $G_v(\phi_1)$ and $G_v(\phi_2)$. But $G_v(\phi)$ has an additional hyperedge from leaves of $G_v(\phi_1)$ to the roots of $G_v(\phi_2)$. The new hyperedge is denoted as (L_1, R_2) below. The set of roots of $G_v(\phi)$ is R_1, while the set of leaves is L_2.

$G_v(\phi) = (V_1 \cup V_2, R_1, L_2, E_1 \cup E_2 \cup \{(L_1, R_2)\}, Lit_1 \cup Lit_2)$

The same idea of using hyperedges applies during the construction of hpgraph. With the use of hyperedges the complexity of obtaining vpgraph/hpgraph is linear in the size of the given NNF formula.

Appendix B

Algorithms from Chapter 3

B.1 Algorithm for Detection of Global conflict

We say that a global conflict occurs when an assignment σ falsifies a given formula ϕ. In order to detect this conflict we use Corollary 3 (Chapter 3). This requires checking if there is an rl-path π in hpgraph $G_h(\phi) = (V', R', L', E', Lit')$ such that σ falsifies π. We present an $O(V' + E')$ algorithm below. We reduce the problem of finding an rl-path π in $G_h(\phi)$ such that σ falsifies π to a shortest path computation problem as follows: It is assumed that σ is consistent, that is, it does not contain opposite literals. For each node $n \in V'$ we assign a weight $w(n) \in \{0, 1, 2\}$ to n. The assignment of weights is done as follows:

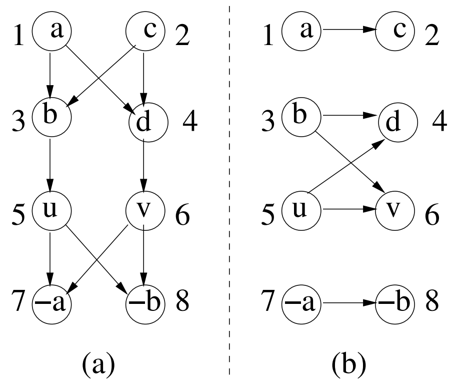


Figure B.1: (a) Vpgraph and (b) Hpgraph for formula $(a \vee c) \wedge ((b \wedge u) \vee (d \wedge v)) \wedge (\neg a \vee \neg b)$.

$$w(n) := \begin{cases} 0 & : \neg Lit(n) \in \sigma \\ 2 & : Lit(n) \in \sigma \\ 1 & : \text{otherwise} \end{cases}$$

We will use the hpgraph in Fig. B.1(b) as our running example. If $\sigma = \{a, b\}$, then the weight assigned to various nodes in the hpgraph is as follows: $w(1) = 2, w(2) = 1, w(3) = 2, w(4) = 1, w(5) = 1, w(6) = 1, w(7) = 0, w(8) = 0$.

Given a path π in hpgraph, we define the length of π to be the sum of weights of the nodes that lie on π. For each node n in the hpgraph we compute a shortest path estimate $\delta(n)$ which represents the length of shortest path from any root node to n. We also track the parent $par(n)$ of each node n in the shortest path to n. For the hpgraph in Fig. B.1(b) and $\sigma = \{a, b\}$, we have $\delta(1) = 2, \delta(2) = 3, \delta(3) = 2, \delta(4) = 2, \delta(5) = 1, \delta(6) = 2, \delta(7) = 0, \delta(8) = 0$.

If there is an rl-path $\pi := \langle n_1, \ldots, n_k \rangle$ in $G_h(\phi)$ such that σ falsifies π, then $\delta(n_k) = 0$. This is because n_1 is a root node (by definition of rl-path) and every node on π has a weight of 0 because σ falsifies each node on π (as σ falsifies π). Thus, there is a path of length 0 to n_k which is the smallest possible length due to non-negative weights. Observe that n_k is a leaf node by definition of a rl-path. The following claim formalizes the above idea of detecting global conflicts using the shortest path estimates.

Claim 1 *The following statements are equivalent:*

1. σ falsifies ϕ.

2. There is an rl-path π in $G_h(\phi)$ such that σ falsifies π.

3. There is a node $n \in L'$ such that $\delta(n) = 0$.

Given a hpgraph and an assignment we compute the shortest path estimates for each node in the hpgraph. If there is a leaf node n in hpgraph such that $\delta(n) = 0$, then there is a global conflict. Otherwise, there is no global conflict. For the hpgraph in Fig. B.1(b) and $\sigma = \{a, b\}$, we have $\delta(8) = 0$ and node 8 is a leaf node, it follows from the above claim that σ falsifies ϕ.

Extraction of falsified rl-path: If there is a a leaf node n in hpgraph such that $\delta(n) = 0$, then the actual rl-path (ending at n) which is falsified by σ can be obtained by examining the parent of each node in the shortest path tree starting from node n. We assume that parent of a root node is `nil` node. Then the required rl-path is obtained by reversing the following sequence $\langle n, par(n), par(par(n)), \ldots, \texttt{nil} \rangle$ and removing the `nil` node.

For the hpgraph in Fig. B.1(b) and $\sigma = \{a, b\}$, we have $\delta(8) = 0$, $par(8) = 7, par(7) = \texttt{nil}$. Thus, the rl-path in hpgraph which is falsified by σ is $\langle 7, 8 \rangle$.

Obtaining unit literals via hpgraph: If there is no leaf node n in hpgraph such that $\delta(n) = 0$, then there is no global conflict. In this case the set of implied assignments under σ can be obtained by applying Corollary 6. More specifically if there is an rl-path $\pi := \langle n_1, \ldots, n_k \rangle$ in $G_h(\phi)$ such that σ falsifies all but one node (say $n_i, 1 \leq i \leq k$) on π and $Lit(n_i)$ is not yet assigned by σ, then $\delta(n_k) = 1$. This is because n_1 is a root node (by definition of rl-path) and every node on π different from n_i has a weight of 0 and n_i has a weight of 1 as $Lit(n_i)$ has not set been assigned by σ. Thus, there is a path of length 1 to n_k which is the smallest possible length given that there is no global conflict, that is, $\delta(n_k) \neq 0$. Observe that n_k is a leaf node by definition of a rl-path. The following claim formalizes the idea of detecting unit literals using the shortest path estimates.

Claim 2 *Assuming σ does not falsify ϕ, the following statements are equivalent:*
1. σ falsifies all but one node (say n) on a rl-path π in $G_h(\phi)$ and $Lit(n)$ is not yet assigned by σ.
2. There is a node $n \in L'$ such that $\delta(n) = 1$.

Using the above claim it is possible to extract various implied literals by examining leaf nodes whose shortest path estimate is 1 (assuming no global conflict). If there is a a leaf node n in hpgraph such that $\delta(n) = 1$, then the actual rl-path π whose all but one node is falsified by current assignment σ can be obtained by examining the parent of each node in the shortest path tree starting from node n. This allows obtaining both the implied literal l and *unit-clause* (literals corresponding to nodes on π) which implied l.

Example 42 For the hpgraph in Fig. B.1(b) and $\sigma = \{a\}$. The shortest path estimates for various nodes are as follows: $\delta(1) = 2, \delta(2) = 3, \delta(3) = 1, \delta(4) = 2, \delta(5) = 1, \delta(6) = 2, \delta(7) = 0, \delta(8) = 1$. Observe that none of the leaf nodes $n \in \{2,4,6,8\}$ has $\delta(n) = 0$. Thus, there is no global conflict. However, $\delta(8) = 1$ which means that there is an rl-path in hpgraph such that σ falsifies all but one node of this rl-path (using above claim). In this example, the required rl-path is $\langle 7,8 \rangle$. Using Corollary 5 $Lit(8) = \neg b$ must be set to true under the current assignment to prevent a global conflict.

Efficiency issues: Since the hpgraph is a DAG the computation of $\delta(n)$ for all n can be done in linear time. However, in practice the routine for detecting global conflicts is called very often, and computing $\delta(n)$ for every (un)assignment to a variable is expensive. Thus, we use two optimizations which are crucial for efficiency: 1) incremental shortest path computation: whenever a variable is (un)assigned, instead of computing the shortest path estimate for every node in the hpgraph, we only examine the nodes whose shortest path estimate can get affected due to this assignment. 2) by limiting the range of $\delta(n)$ to only $\{0, 1, \infty\}$.

B.2 Algorithm for Detection of Local Conflict

A local conflict occurs when every rl-path in $G_v(\phi)$ with `CRP`$\langle m \rangle$ as prefix contains two nodes which are conflicting and one of the conflicting nodes lies on

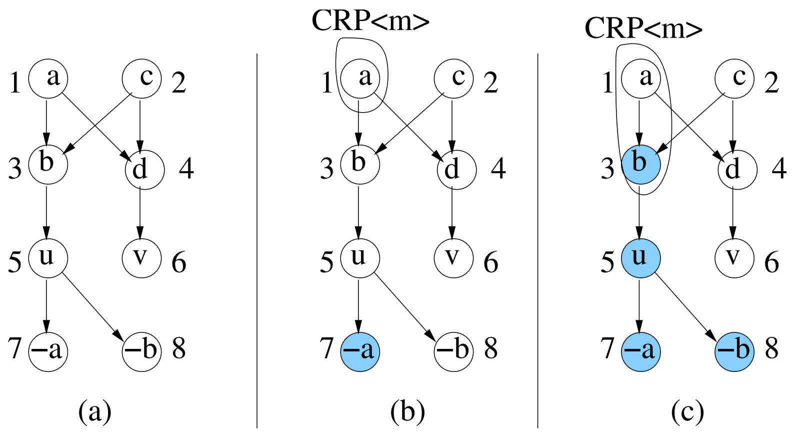


Figure B.2: (a) Vpgraph for formula $(a \vee c) \wedge ((b \wedge u \wedge (\neg a \vee \neg b)) \vee (d \wedge v))$ (b) assignment of conflict labels when CRP is $\langle\rangle$ and $m = 1$. A colored node n denotes $conf(n)$ is true. (c) assignment of conflict labels when CRP is $\langle 1 \rangle$ and $m = 3$. A local conflict occurs as $conf(3)$ is true.

CRP$\langle m \rangle$. This conflict can be detected by using a linear time algorithm as described below.

Let σ denote the set of literals corresponding to the nodes on CRP$\langle m \rangle$. That is, $\sigma = \{Lit(n) | n \in \text{CRP}\langle m \rangle\}$. In order to detect a local conflict, we compute for each node n in the vpgraph a flag $conf(n)$. If $conf(n)$ is true for some node n, then no satisfiable rl-path with CRP$\langle m \rangle$ as prefix can pass through this node. A local conflict happens when $conf(m)$ becomes true.

We compute $conf(n)$ for every n by scanning the nodes in $G_v(\phi)$ in reverse topological order (recall that $G_v(\phi)$ is a DAG). For each node n we assign $conf(n)$ as follows:

1. If $\neg Lit(n) \in \sigma$, then set $conf(n)$ to true.
2. Else if $conf(n')$ is true for every successor n' of n, then set $conf(n)$ to true.

3. Else set $conf(n)$ to false.

Example 43 Consider the vpgraph in Fig. B.2(a). Suppose `CRP` is $\langle\rangle$ and $m = 1$. Using the above notation $\sigma = \{a\}$. The assignment of conflict labels sets $conf(7)$ to true and $conf(n)$ to false for all other n as shown in Fig. B.2(b). Since $conf(1)$ is false, there is no local conflict when extending `CRP` by node 1. Now consider the case when `CRP` is $\langle 1\rangle$ and m is node 3. In this case $\sigma = \{a,b\}$. The assignment of conflict labels sets $conf(n) = true$ for $n \in \{3,5,7,8\}$ to true and $conf(n)$ to false for all other n as shown in Fig. B.2(c). Since $conf(3)$ is true, there is a local conflict when extending `CRP` by node 3.

Efficiency issues: In our implementation we do not carry out the above computation of scanning the entire vpgraph whenever a variable is (un)assigned. Instead, we incrementally update the $conf(n)$ flags by remembering them across multiple calls to the local conflict detection routine. Furthermore, our algorithm only looks at the nodes whose $conf(n)$ flag may get affected due to a variable (un)assignment.

Appendix C

Proof of Duality Between rl-cuts and rl-paths

Given two paths $\pi_1 := \langle m_1, \ldots, m_k \rangle$ and $\pi_2 := \langle m_{k+1}, \ldots, m_t \rangle$, we use $\pi_1.\pi_2$ to denote the path obtained by concatenating π_1 with π_2, that is, $\langle m_1, \ldots, m_k, m_{k+1}, \ldots, m_t \rangle$. In order to prove theorems 6,7 we make use of the following lemmas.

Lemma 8 *Given $\phi = \phi_1 \vee \phi_2$. The following are equivalent:*
(a) C is a minimal rl-cut in $G_h(\phi_1)$ or $G_h(\phi_2)$
(b) C is a minimal rl-cut in $G_h(\phi)$.

Proof. Let $G_h(\phi_1) = (V_1, R_1, L_1, E_1, Lit_1)$ and $G_h(\phi_2) = (V_2, R_2, L_2, E_2, Lit_2)$. The hpgraph of ϕ is obtained by connecting the leafs in the hpgraph of ϕ_1 with roots in the hpgraph of ϕ (see Figure C.1). Formally, $G_h(\phi) = (V_1 \cup V_2, R_1, L_2, E_1 \cup E_2 \cup (L_1 \times R_2), Lit_1 \cup Lit_2)$.

(a) $\Rightarrow$ (b): We consider two cases:

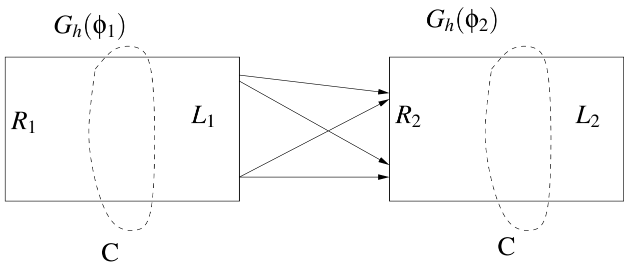


Figure C.1: Hpgraph of $\phi_1 \vee \phi_2$ is obtained by connecting hpgraph of ϕ_1 with the hpgraph of ϕ_2.

- C is a minimal rl-cut in $G_h(\phi_1)$. Every rl-path π in $G_h(\phi)$ is of the form $\pi_1.\pi_2$, where π_1 is a rl-path in $G_h(\phi_1)$ and π_2 is a rl-path in $G_h(\phi_2)$. Observe that C will also be a rl-cut for $G_h(\phi)$ as every rl-path in $G_h(\phi)$ will be disconnected by removing nodes from C. Suppose C is not a minimal rl-cut in $G_h(\phi)$, then there must exist $C' \subset C$ such that C' is a minimal rl-cut. But this means that C' is also a minimal rl-cut for $G_h(\phi_1)$ leading to a contradiction. Thus, C is a minimal rl-cut in $G_h(\phi)$.

- C is a minimal rl-cut in $G_h(\phi_2)$. We can use similar reasoning as above to prove that C is a minimal rl-cut in $G_h(\phi)$.

(b) $\Rightarrow$ (a): We consider three cases.

- $C \subseteq V_1$. It is easy to see that C is a minimal rl-cut in $G_h(\phi_1)$.

- $C \subseteq V_2$. It is easy to see that C is a minimal rl-cut in $G_h(\phi_2)$.

- $C = C_1 \cup C_2, C_1 \subseteq V_1, C_2 \subseteq V_2, C_1 \neq \emptyset, C_2 \neq \emptyset$. We show that this case cannot arise by using proof by contradiction. Observe that C_1 cannot be a rl-cut in

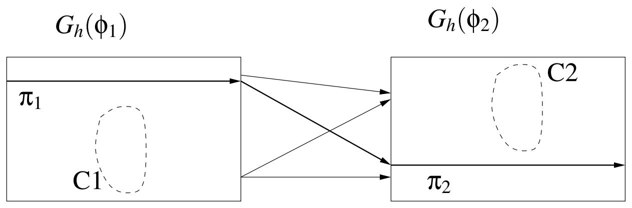


Figure C.2: Hpgraph for $\phi_1 \vee \phi_2$.

$G_h(\phi_1)$ (otherwise $C_1 \cup C_2$ cannot be a minimal rl-cut in $G_h(\phi)$). Similarly, C_2 cannot be a rl-cut in $G_h(\phi_2)$. Thus, there exists a rl-path π_1 in $G_h(\phi_1)$ which does not contain any node from C_1. Similarly, there exists a rl-path π_2 in $G_h(\phi_2)$ which does not contain any node from C_2. The rl-path $\pi_1.\pi_2$ belongs to $G_h(\phi)$ and does not contain any node from $C_1 \cup C_2$ (see Figure C.2). This means that $C_1 \cup C_2$ cannot be a rl-cut for $G_h(\phi)$ leading to a contradiction.

□

Lemma 9 *Given $\phi = \phi_1 \wedge \phi_2$. Let $G_h(\phi_1) = (V_1, R_1, L_1, E_1, Lit_1)$ and $G_h(\phi_2) = (V_2, R_2, L_2, E_2, Lit_2)$. Then $G_h(\phi)$ is obtained by taking a union of $G_h(\phi_1)$ and $G_h(\phi_2)$. That is, $G_h(\phi) = (V_1 \cup V_2, R_1 \cup R_2, L_1 \cup L_2, E_1 \cup E_2, Lit_1 \cup Lit_2)$. The following are equivalent:*
(a) C_1 is a minimal rl-cut in $G_h(\phi_1)$ and C_2 is a minimal rl-cut in $G_h(\phi_2)$.
(b) $C_1 \cup C_2$ is a minimal rl-cut in $G_h(\phi)$, where $C_1 \subseteq V_1, C_2 \subseteq V_2$.

Proof. (a) $\Rightarrow$ (b): Note that each rl-path in $G_h(\phi)$ is a rl-path in either $G_h(\phi_1)$ or $G_h(\phi_2)$. Thus, $C_1 \cup C_2$ is a rl-cut in $G_h(\phi)$. It is also easy to see that $C_1 \cup C_2$ is a

minimal rl-cut in $G_h(\phi)$. Otherwise, we can show that either C_1 is not a minimal rl-cut in $G_h(\phi_1)$ or C_2 is not a minimal rl-cut in $G_h(\phi_2)$.
(b) $\Rightarrow$ (a): It is easy to see that C_1 will be a rl-cut in $G_h(\phi_1)$ and C_2 will be a rl-cut in $G_h(\phi_2)$. Since $C_1 \cup C_2$ is a minimal rl-cut in $G_h(\phi)$ it follows that both C_1, C_2 are minimal rl-cuts in $G_h(\phi_1)$, $G_h(\phi_2)$, respectively. $\square$

Theorem 6. *Given hpgraph $G_h(\phi)$ and vpgraph $G_v(\phi)$ for a formula ϕ. Let π be a rl-path in $G_v(\phi)$. Then $nodes(\pi)$ form a minimal rl-cut in $G_h(\phi)$.*

Proof. We prove this theorem by induction on the structure of ϕ.

- ϕ is a literal l: In this case both $G_h(\phi)$ and $G_v(\phi)$ contain a single node n with $Lit(n) = l$. In this case the rl-path in $G_v(\phi)$ is simply $\langle n \rangle$ and $\{n\}$ is a rl-cut for $G_h(\phi)$.

- $\phi = \phi_1 \vee \phi_2$. Since $G_v(\phi)$ is obtained by taking a union of $G_v(\phi_1)$ and $G_v(\phi_2)$ we have two cases: 1) π is a rl-path in $G_v(\phi_1)$. By induction hypothesis $nodes(\pi)$ form a minimal rl-cut in $G_h(\phi_1)$. From lemma 8 it follows that $nodes(\pi)$ form a minimal cut in $G_h(\phi)$. 2) π is a rl-path in $G_v(\phi_2)$. We can use similar reasoning as case 1 to conclude that $nodes(\pi)$ form a minimal cut in $G_h(\phi)$.

- $\phi = \phi_1 \wedge \phi_2$. Since $G_v(\phi)$ is obtained by connecting leafs in $G_v(\phi_1)$ with the roots in $G_v(\phi_2)$, $\pi = \pi_1.\pi_2$ where π_1 is a rl-path in $G_v(\phi_1)$ and π_2 is a rl-path in $G_v(\phi_2)$. By induction hypothesis $nodes(\pi_1)$ form a minimal rl-cut in in

$G_h(\pi_1)$ and $nodes(\pi_2)$ form a minimal rl-cut in in $G_h(\pi_2)$. From lemma 9 it follows that $nodes(\pi_1) \cup nodes(\pi_2) = nodes(\pi)$ form a minimal cut in $G_h(\phi)$.

□

Theorem 7. *Given hpgraph $G_h(\phi)$ and vpgraph $G_v(\phi)$ for a formula ϕ. Let C be a minimal rl-cut in $G_h(\phi)$. Then there exists a rl-path π in $G_v(\phi)$ such that $C = nodes(\pi)$.*

Proof. We prove this theorem by induction on the structure of ϕ.

- ϕ is a literal l: In this case both $G_h(\phi)$ and $G_v(\phi)$ contain a single node n with $Lit(n) = l$. In this case $C = \{n\}$ and $\pi = \langle n \rangle$.
- $\phi = \phi_1 \vee \phi_2$. From lemma 8 we have two possible cases. 1) C is a minimal cut in $G_h(\phi_1)$. By induction hypothesis there exists a rl-path π in $G_v(\phi_1)$ such that $C = nodes(\pi)$. Since $G_v(\phi)$ is obtained by taking a union of $G_v(\phi_1)$ and $G_v(\phi_2)$ it follows that π is a rl-path in $G_v(\phi)$. 2) C is a minimal cut in $G_h(\phi_2)$. Using similar reasoning as case 1 we can argue that there exists a rl-path π in $G_v(\phi)$ such that $C = nodes(\pi)$.
- $\phi = \phi_1 \wedge \phi_2$. From lemma 9 it follows that $C = C_1 \cup C_2$ where C_1 is a minimal rl-cut in $G_h(\phi_1)$ and C_2 is a minimal rl-cut in $G_h(\phi_2)$. By induction hypothesis there exists a rl-path π_1 in $G_v(\phi_1)$ such that $C_1 = nodes(\pi_1)$ and

there exists a rl-path π_2 in $G_v(\phi_2)$ such that $C_2 = nodes(\pi_2)$. But $\pi = \pi_1.\pi_2$ is a rl-path in $G_v(\phi)$ and $C = C_1 \cup C_2 = nodes(\pi_1) \cup nodes(\pi_2) = nodes(\pi)$.

$\square$

Appendix D

Proofs from Chapter 6

D.1 Proofs from Section 6.4

Proof of Lemma 1

Proof. $UCX = UD$ is a linear combination of equations in $CX = D$. Let X_0 be an integral solution to $CX = D$. It is easy to verify that X_0 also satisfies $UCX = UD$. Thus, the system of LDEs $CX = D$ implies the LDE $UCX = UD$ for any rational row vector U.

Since $UCX_0 - UD = 0$, any rational number m divides $UCX_0 - UD$. It follows that X_0 is also a solution to the LME $UCX \equiv_m UD$. Thus, the system of LDEs $CX = D$ implies the LME $UCX \equiv_m UD$ for any rational row vector U and rational number m. $\square$

Why $F \wedge G$ has no LDE as interpolant in Example 5.

Proof. Recall, that F is $x - 2y = 0$ and G is $x - 2z = 1$, where x, y, z are integers. Observe that F has an integral solution, for example, $x = 2, y = 1$. Thus, by lemma 6 any LDE that is implied by F must be of the form $r(x - 2y = 0)$, where r is a rational number.

Suppose (F, G) have an LDE I as an interpolant. Since $F \Rightarrow I$, I must be of the form $r(x - 2y = 0)$. But I can only contain variable x (common variable of F and G). This is possible only when $r = 0$. With $r = 0$, I reduces to $0 = 0$ which is not unsatisfiable with G. Thus, (F, G) cannot have an LDE as an interpolant. □

Proof of Lemma 2

Proof. By definition of $V_{A \setminus B}$ the coefficient of $x_i \in V_{A \setminus B}$ is zero in each equation of $BX = B'$. Thus, the coefficient of $x_i \in V_{A \setminus B}$ must be the same in R_1AX and $(R_1A + R_2B)X$. Since $R_1A + R_2B$ is integral it follows that the coefficient of $x_i \in V_{A \setminus B}$ (a_i) in the partial interpolant is an integer. □

D.1.1 Proof of Lemma 3

Lemma 3. *The partial interpolant $R_1AX = R_1A'$ satisfies the first two conditions in the definition of an interpolant. That is,*

1. $AX = A'$ implies $R_1AX = R_1A'$

2. $(R_1AX = R_1A') \wedge BX = B'$ is unsatisfiable

If $a_i = 0$ for all $x_i \in V_{A\setminus B}$ (equation 6.1), then the partial interpolant is also a interpolant for $(AX = B, A'X = B')$. In this case the partial interpolant only contains the variables from V_{AB}.

Proof. 1. $AX = A'$ implies $R_1AX = R_1A'$. This follows from Lemma 1.

2. Observe that $(R_1AX = R_1A') \wedge BX = B'$ is a system of LDEs

$$\begin{bmatrix} R_1A \\ B \end{bmatrix} X = \begin{bmatrix} R_1A' \\ B' \end{bmatrix}$$

We show that the row vector $[1, R_2]$ is a proof of unsatisfiability of $I \wedge (BX = B')$. This requires showing the conditions in the definition of proof of unsatisfiability are met.

- To show

$$[1, R_2] \begin{bmatrix} R_1A \\ B \end{bmatrix} \text{ is integral.}$$

The above product is equal to $R_1A + R_2B$ which is integral.

- To show

$$[1, R_2] \begin{bmatrix} R_1A' \\ B' \end{bmatrix} \text{ is not an integer.}$$

The above product is equal to $R_1A' + R_2B'$ which is not an integer. Thus, $[1, R_2]$ is a proof of unsatisfiability of $I \wedge (BX = B')$. So $I \wedge (BX = B')$ is unsatisfiable. $\square$

D.1.2 Proof of Theorem 12

Recall that rational row vector $[R_1, R_2]$ is the proof of unsatisfiability of $AX = A' \wedge BX = B'$ (A, B, A', B' are rational matrices) such that

$$\begin{array}{rl} R_1A + R_2B & \text{is integral} \\ R_1A' + R_2B' & \text{is not an integer} \end{array}$$

We call $R_1AX = R_1A'$ the partial interpolant for $(AX = A', BX = B')$. It can be written as follows:

$$\sum_{x_i \in V_{A \setminus B}} a_i x_i + \sum_{x_i \in V_{AB}} b_i x_i = c \qquad \text{(D.1)}$$

where all coefficients a_i, b_i and $c = R_1A'$ are rational numbers. The above equation is the same as Equation 6.1 repeated here for convenience.

Similarly, $R_2BX = R_2B'$ can be written as follows:

$$\sum_{x_i \in V_{AB}} e_i x_i + \sum_{x_i \in V_{B \setminus A}} f_i x_i = d \qquad \text{(D.2)}$$

where all coefficients e_i, f_i and $d = R_2B'$ are rational numbers. Observe that $R_2BX = R_2B'$ does not contain any variable from $V_{A \setminus B}$.

Lemma 10 *Using the notation from Equations D.1 and D.2:*

(a) For all $x_i \in V_{A \setminus B}$, a_i is an integer.

(b) For all $x_i \in V_{AB}$, $b_i + e_i$ is an integer.

(c) For all $x_i \in V_{B \setminus A}$, f_i is an integer.

(d) $c + d$ is not an integer.

Proof. The sum of the left hand sides of Equations D.1 and D.2 is

$$\sum_{x_i \in V_{A \setminus B}} a_i x_i + \sum_{x_i \in V_{AB}} (b_i + e_i) x_i + \sum_{x_i \in V_{B \setminus A}} f_i x_i$$

which is the same as $(R_1 A + R_2 B)X$. Since $R_1 A + R_2 B$ is integral each coefficient in the above sum must be an integer. This gives us the desired results (a),(b),(c).

Since $c + d = R_1 A' + R_2 B'$ and $R_1 A' + R_2 B'$ is not an integer we get (d). $\square$

Theorem 12. *Assume that the coefficient a_i of at least one $x_i \in V_{A \setminus B}$ in the partial interpolant (Equation D.1) is not zero. Let α denote the gcd of $\{a_i | x_i \in V_{A \setminus B}\}$.*

(a) α is an integer and $\alpha > 0$.

(b) Let β be any integer that divides α. Then the following linear modular equation I_β is an interpolant for $(AX = A', BX = B')$.

$$I_\beta := \sum_{x_i \in V_{AB}} b_i x_i \equiv c \ (mod \ \beta)$$

Observe that I_β contains only variables that are common to both $AX = A'$ and $BX = B'$. It is obtained from the partial interpolant (Equation D.1) by dropping *all variables occurring only in $AX = A'$ ($V_{A \setminus B}$) and replacing the linear equality by a modular equality.*

Proof. (a) By lemma 10 each a_i is an integer. Since α is the gcd of $\{a_i | x_i \in V_{A \setminus B}\}$, α must be an integer. Also note that α is non-zero since at least one a_i is non-zero. By definition of gcd α is positive.

(b) To show that I_β is an interpolant for $(AX = A', BX = B')$.

1. We need to show that $AX = A'$ implies I_β. Recall, that $AX = A'$ implies the partial interpolant $R_1 AX = R_1 A'$ from lemma 3. We show that $R_1 AX = R_1 A'$ implies I_β.

From basic modular arithmetic it follows that $s = t$ implies $s \equiv t \ (mod\ \gamma)$ for any rational number γ. Thus, the partial interpolant $R_1 AX = R_1 A'$ implies $R_1 AX \equiv_\beta R_1 A'$, where β is any integer that divides α. Consider the equation form of $R_1 AX \equiv_\beta R_1 A'$ (equation D.1):

$$\sum_{x_i \in V_{A \setminus B}} a_i x_i + \sum_{x_i \in V_{AB}} b_i x_i \equiv_\beta c \tag{D.3}$$

By definition α divides a_i for all $x_i \in V_{A \setminus B}$. Since β divides α, it follows that β divides a_i for all $x_i \in V_{A \setminus B}$. As x_i is an integer valued variable, $a_i x_i$ is divisible by β for all $x_i \in V_{A \setminus B}$. It follows that

$$\sum_{x_i \in V_{A \setminus B}} a_i x_i \equiv_\beta 0. \tag{D.4}$$

Subtract equation D.4 from equation D.3 to obtain

$$\sum_{x_i \in V_{AB}} b_i x_i \equiv_\beta c.$$

The above equation is I_β. $AX = A'$ implies $R_1AX = R_1A'$ and $R_1AX = R_1A'$ implies equation D.3. Equation D.4 holds for any integral assignment to all $x_i \in V_{A\setminus B}$. So $R_1AX = R_1A'$ implies equation D.4. Equations D.3, D.4 imply I_β. It follows that $AX = A'$ implies I_β.

2. We need to show that $I_\beta \wedge BX = B'$ is unsatisfiable. Assume for the sake of contradiction that $I_\beta \wedge BX = B'$ has an integral satisfying assignment. Let the satisfying assignment to $I_\beta \wedge BX = B'$ be $x_i = g_i$ where g_i is an integer for all $x_i \in V_{AB} \cup V_{B\setminus A}$. Since I_β is satisfied by g_i we have

$$\sum_{x_i \in V_{AB}} b_i g_i \equiv_\beta c$$

Thus, there exists an integer t such that

$$\sum_{x_i \in V_{AB}} b_i g_i + t\beta = c \tag{D.5}$$

The equation $R_2BX = R_2B'$ is implied by $BX = B'$. Thus, the satisfying assignment $x_i = g_i$ for all $x_i \in V_{AB} \cup V_{B\setminus A}$ satisfies $R_2BX = R_2B'$. By plugging in the values g_i

for x_i in Equation D.2 we get:

$$\sum_{x_i \in V_{AB}} e_i g_i + \sum_{x_i \in V_{B \setminus A}} f_i g_i = d \tag{D.6}$$

We can sum the equations D.5, D.6 to get

$$t\beta + \sum_{x_i \in V_{AB}} (b_i + e_i) g_i + \sum_{x_i \in V_{B \setminus A}} f_i g_i = c + d \tag{D.7}$$

We know that t, β are integers, g_i are integers for all $x_i \in V_{AB} \cup V_{B \setminus A}$, and from Lemma 10 it follows that $b_i + e_i$ is integer for $x_i \in V_{AB}$ and f_i is integer for $x_i \in V_{B \setminus A}$. It follows that the left hand side of Equation D.7 is an integer. While the right hand side of Equation D.7 is not an integer by Lemma 10. Thus, the above equation is the required contradiction. It follows that $I_\beta \wedge BX = B'$ are unsatisfiable.

3. By the definition of I_β it follows that I_β only contains common variables of $AX = A'$ and $BX = B'$. $\square$

D.2 Proofs from Section 6.5

D.2.1 Proof of Theorem 13

In order to prove theorem 13 we reduce the given system of LMEs to an equisatisfiable system of LDEs. We then use theorem 11 about the satisfiability of LDEs in order to complete the proof.

Reduction of a System of LMEs to an Equisatisfiable System of LDEs

Suppose we are given a system $CX \equiv_l D$ of linear modular equations:

$$\underbrace{\begin{bmatrix} c_{11} & \dots & c_{1n} \\ c_{21} & \dots & c_{2n} \\ \dots & & \\ c_{m1} & \dots & c_{mn} \end{bmatrix}}_{C} \underbrace{\begin{bmatrix} x_1 \\ . \\ . \\ x_n \end{bmatrix}}_{X} \equiv_l \underbrace{\begin{bmatrix} d_1 \\ d_2 \\ . \\ d_m \end{bmatrix}}_{D}$$

For each equation $\sum_j c_{ij}x_j \equiv_l d_i$ in $CX \equiv_l D$ we introduce a new **integer** variable v_i, to obtain a new equation (without modulo), given as follows:

$$\sum_{j=1}^{n} c_{ij}x_j + lv_i = d_i$$

The above equation is equi-satisfiable to the linear modular equation $\sum_j c_{ij}x_j \equiv_l d_i$. Let V denote the vector of variables $v_1, \dots, v_m$. We call the new system of linear equations as $C'Z = D$, where Z denotes the concatenation of variable vectors X and V. Note that $C'Z = D$ is a system of linear diophantine equations.

$$\underbrace{\begin{bmatrix} c_{11} & \dots & c_{1n} & l & 0 & \dots & 0 \\ c_{21} & \dots & c_{2n} & 0 & l & \dots & 0 \\ \dots & & & & & & \\ c_{m1} & \dots & c_{mn} & 0 & 0 & \dots & l \end{bmatrix}}_{C'} \underbrace{\begin{bmatrix} x_1 \\ . \\ x_n \\ v_1 \\ . \\ v_m \end{bmatrix}}_{Z} = \underbrace{\begin{bmatrix} d_1 \\ . \\ . \\ d_m \end{bmatrix}}_{D}$$

Lemma 11 *The following are equivalent:*

(a) the system of linear modular equations $CX \equiv_l D$ has an integral solution

(b) the system of linear diophantine equations $C'Z = D$ has an integral solution.

Proof. The proof of the above lemma is elementary.

Theorem 13. *Let C be a rational matrix, D be a rational column vector, and l be a rational number. The system $CX \equiv_l D$ has no integral solution X if and only if there exists a rational row vector R such that RC is integral, lR is integral, and RD is not an integer.*

From lemma 11 and theorem 11 the following are equivalent:

(a) linear modular equations $CX \equiv_l D$ has no integral solution

(b) linear diophantine equations $C'Z = D$ has no integral solution

(c) There exists a row vector R such that RC' is integral and RD is not an integer.

We show that the property of R in (c) is equivalent to "(d) RC is integral, lR is integral, and RD is not an integer".

Let $R = [r_1, \ldots, r_m]$ then

$$RC' = \left[\sum_{i=1}^{m} r_i c_{i1}, \sum_{i=1}^{m} r_i c_{i2}, \ldots, \sum_{i=1}^{m} r_i c_{in},\ lr_1, \ldots, lr_i, \ldots, lr_m\right]$$

$$RC' = [RC, lR]$$

Thus, RC' is integral if and only if RC and lR are integral. This shows (c) is equivalent to (d). Thus, (a) is equivalent to (d) as required by the proof. $\square$

D.2.2 Proof of Theorem 14

Recall that $V_{A\setminus B}$ denotes the set of variables that occur **only** in $AX \equiv_l A'$ (and not in $BX \equiv_l B'$) and V_{AB} denotes the set of variables that occur in both $AX \equiv_l A'$ and $BX \equiv_l B'$. The rational row vector $R = [R_1, R_2]$ is a proof of unsatisfiability of $AX \equiv_l A' \wedge BX \equiv_l B'$ such that

$$R_1 A + R_2 B \qquad \text{is integral} \tag{D.8}$$

$$lR = [lR_1, lR_2] \qquad \text{is integral} \tag{D.9}$$

$$R_1A' + R_2B' \qquad \text{is not an integer.} \tag{D.10}$$

Lemma 12 *The coefficient of $x_i \in V_{A \setminus B}$ in R_1AX is an integer.*

Proof. By definition of $V_{A\setminus B}$ the coefficient of $x_i \in V_{A\setminus B}$ is zero in R_2BX. Thus, the coefficient of $x_i \in V_{A\setminus B}$ is the same in R_1AX and $(R_1A + R_2B)X$. We know $R_1A + R_2B$ is integral from equation D.8. So the coefficient of $x_i \in V_{A\setminus B}$ in R_1AX is an integer. $\square$

Theorem 14. *We assume $l \neq 0$. Let S_1 denote the set of non-zero coefficients of $x_i \in V_{A\setminus B}$ in R_1AX. Let S_2 denote the set of all non-zero elements of row vector lR_1. If $S_2 = \emptyset$, then the interpolant for $(AX \equiv_l A', BX \equiv_l B')$ is a trivial LME $0 \equiv_l 0$. Otherwise, let $S_2 \neq \emptyset$. Let α denote the gcd of numbers in $S_1 \cup S_2$. (a) α is an integer and $\alpha > 0$. (b) Let β be any integer that divides α. Let $U = \frac{l}{\beta}R_1$. Then $UAX \equiv_l UA'$ is an interpolant for $(AX \equiv_l A', BX \equiv_l B')$.*

Proof. $S_2 = \emptyset$**:** If $S_2 = \emptyset$ it follows that all elements of lR_1 are zero. Since $l \neq 0$, R_1 must be a zero vector. It follows that R_1A is a zero vector and $R_1A' = 0$. Using equation D.8 and R_1A is a zero vector, it follows that R_2B is integral. Using equation D.10 and $R_1A' = 0$, it follows that R_2B' is not an integer. Thus, $BX \equiv_l B'$ is itself unsatisfiable with R_2 as the proof of unsatisfiability. In this case we can simply take `true` as the interpolant for the pair $(AX \equiv_l A', BX \equiv_l B')$. The interpolant `true` can be expressed as a trivial LME $0 \equiv_l 0$.

$S_2 \neq \emptyset$: We first show that α is an integer. Since lR_1 is integral (see equation D.9) all elements of S_2 are non-zero integers. All elements of S_1 are non-zero integers due to Lemma 12. Thus, $S_1 \cup S_2$ is a set of non-zero integers. Since $S_2 \neq \emptyset$ there exists at least one element in $S_1 \cup S_2$. α is the gcd of the numbers in $S_1 \cup S_2$. So α is a non-zero integer and by definition of gcd α is positive.

Let β be any integer that divides α. Note that $\beta \neq 0$ as $\alpha \neq 0$. We define

$$I_\beta := UAX \equiv_l UA' \qquad \text{where} \qquad U = \frac{l}{\beta} R_1. \tag{D.11}$$

We need to show that I_β is an interpolant for the pair $(AX \equiv_l A', BX \equiv_l B')$.

(a) To show $AX \equiv_l A' \Rightarrow I_\beta$. If we show that U is integral, then by lemma 4 it follows that $AX \equiv_l A' \Rightarrow UAX \equiv_l UA'$ and thus $AX \equiv_l A' \Rightarrow I_\beta$. We need to show that U is integral.

Recall from equation D.9 that lR_1 is integral. By definition of α it follows that α divides every element in S_2 or the row vector lR_1. Since β divides α, β divides every element in lR_1. So $\frac{lR_1}{\beta} = \frac{l}{\beta} R_1 = U$ is an integral vector.

(b) To show $I_\beta \wedge (BX \equiv_l B')$ is unsatisfiable. Observe that $I_\beta \wedge (BX \equiv_l B')$ is another system of LMEs

$$\begin{bmatrix} UA \\ B \end{bmatrix} X \equiv_l \begin{bmatrix} UA' \\ B' \end{bmatrix}$$

We show that the row vector $[\frac{\beta}{l}, R_2]$ serves as the proof of unsatisfiability of $I_\beta \wedge (BX \equiv_l B')$. We will check the conditions in the definition of proof of unsatisfiability.

- To show

$$[\frac{\beta}{l}, R_2] \begin{bmatrix} UA \\ B \end{bmatrix} \quad \text{is integral}$$

The above product is equal to $\frac{\beta}{l}(UA) + R_2 B = R_1 A + R_2 B$. By equation D.8 we know that $R_1 A + R_2 B$ is integral.

- To show that $l[\frac{\beta}{l}, R_2] = [\beta, lR_2]$ is integral. From equation D.9, lR_2 is integral and β is an integer by definition.

- To show

$$[\frac{\beta}{l}, R_2] \begin{bmatrix} UA' \\ B' \end{bmatrix} \quad \text{is not an integer}$$

The above product is equal to $\frac{\beta}{l}(UA') + R_2 B' = R_1 A' + R_2 B'$. By equation D.10 we know that $R_1 A' + R_2 B'$ is not an integer.

We conclude that $[\frac{\beta}{l}, R_2]$ is a proof of unsatisfiability of $I_\beta \wedge (BX \equiv_l B')$. Thus, $I_\beta \wedge (BX \equiv_l B')$ is unsatisfiable.

(c) To show that I_β only contains variables that are common to both $(AX \equiv_l A', BX \equiv_l B')$. Since I_β is obtained by a linear combination of equations from $AX \equiv_l A'$, we can write I_β as follows:

$$\underbrace{\sum_{x_i \in V_{A \setminus B}} a_i x_i + \sum_{x_i \in V_{AB}} b_i x_i}_{UAX} \equiv_l \underbrace{c}_{UA'} \tag{D.12}$$

where all coefficients a_i, b_i and $c = UA'$ are rational numbers.

We will show that the coefficient a_i of each $x_i \in V_{A \setminus B}$ in equation D.12 is divisible by l. This will in turn show that

$$\sum_{x_i \in V_{A \setminus B}} a_i x_i \equiv_l 0 \tag{D.13}$$

since x_i are integer variables. This will allow I_β to be written in an equivalent manner (containing only variables from V_{AB}) as follows:

$$\sum_{x_i \in V_{AB}} b_i x_i \equiv_l c.$$

We now show that the coefficient a_i of each $x_i \in V_{A\setminus B}$ in equation D.12 is divisible by l. Recall, that

$$I_\beta := UAX \equiv_l UA' \qquad \text{where} \qquad U = \frac{l}{\beta} R_1 \text{ and } \beta \text{ divides } \alpha. \tag{D.14}$$

By definition α divides every element in S_1
$\Rightarrow$ α divides the coefficient of each $x_i \in V_{A\setminus B}$ in $R_1 AX$
$\Rightarrow$ β divides the coefficient of each $x_i \in V_{A\setminus B}$ in $R_1 AX$.
$\Rightarrow$ the coefficient of $x_i \in V_{A\setminus B}$ in $\frac{1}{\beta} R_1 AX$ is an integer.
$\Rightarrow$ the coefficient of $x_i \in V_{A\setminus B}$ in $l \times \frac{1}{\beta} R_1 AX$ is divisible by l.
$\Rightarrow$ the coefficient of $x_i \in V_{A\setminus B}$ in UAX is divisible by l (as $U = \frac{l}{\beta} R_1$)
The coefficient of $x_i \in V_{A\setminus B}$ in UAX is simply a_i (equation D.12). So l divides a_i.
□

Degenerate case $l = 0$. Let $AX \equiv_l A'$ be a system of LMEs. For $l = 0$, $AX \equiv_l A'$ is equivalent to a system of LDEs $AX = A'$. In order to see this, consider an LME $\sum_{i=1}^{n} a_i x_i \equiv_0 b$. This LME is satisfied if and only if $\sum_{i=1}^{n} a_i x_i - b = 0 \times \lambda$, for some integer λ. Thus, the LME $\sum_{i=1}^{n} a_i x_i \equiv_0 b$ is equivalent to the LDE $\sum_{i=1}^{n} a_i x_i = b$.

Suppose $AX \equiv_0 A' \wedge BX \equiv_0 B'$ is unsatisfiable. Then the interpolant for $(AX \equiv_0 A', BX \equiv_0 B')$ can be obtained by computing the interpolant for the pair of LDEs $(AX = A', BX = B')$.

D.3 Proof of Corollary 14

Corollary 14. *Given $CX = D$ where C, D are rational matrices, and C has full row rank. Let $[E\ \ 0]$ denote the Hermite normal form (HNF) of C. If $CX = D$ has no integral solution, then $E^{-1}D$ is not integral (due to lemma 5). Suppose the i^{th} entry in $E^{-1}D$ is not an integer. Let R' denote the i^{th} row in E^{-1}. Then*
(a) $R'D$ is not an integer
(b) $R'C$ is integral
Thus, R' serves as the required proof of unsatisfiability of $CX = D$.

Proof. (a) Follows from the definition of R'
(b) We know that

$$CU = [E\ \ 0]$$

where U is a unimodular matrix. Since E is invertible (by definition of HNF) we can multiply both sides of the above equation by E^{-1} to obtain

$$E^{-1}CU = E^{-1}[E\ \ 0].$$

The above equation simplifies to

$$E^{-1}CU = [I\ \ 0]$$

where I is the identity matrix. Since U is unimodular its inverse (U^{-1}) exists and it is a unimodular matrix. Multiply both sides of the above equation by U^{-1} to obtain

$$E^{-1}CUU^{-1} = [I\ \ 0]U^{-1}.$$

The above equation simplifies to

$$E^{-1}C = [I\ \ 0]U^{-1}.$$

Since U^{-1} is unimodular the right hand side of the above equation has integral entries. Thus, the left hand side $E^{-1}C$ is integral. In particular the ith row in $E^{-1}C$ is integral. Observe that the ith row in $E^{-1}C$ is simply $R'C$. Thus, $R'C$ is integral. □

D.4 Proof of Lemma 6

We need to introduce cutting-plane proof system [129, 44] in order to prove this lemma. Suppose we are given a system of integer linear inequalities $AX \leq B$, where A, B are rational matrices and X is a column vector of integer variables. The following inference rules allow us to derive new inequalities that are implied by $AX \leq B$.

`nonneg_lin_comb`: We can take a non-negative linear combination of inequalities to derive a new inequality.

$$\frac{AX \leq B}{RAX \leq RB} \qquad R \geq 0$$

(R is a rational row vector whose each element is non-negative.)

`rounding`: If we have a linear inequality $EX \leq F$ such that all coefficients in E are integers ($E \in \mathbb{Z}^n$), then we can round down the right hand side F.

$$\frac{EX \leq F}{EX \leq \lfloor F \rfloor} \qquad E \in \mathbb{Z}^n$$

($EX \leq F$ in the above rule represents a single inequality and not a system of inequalities. E is a row vector containing n integers.) We say an application of the `rounding` rule is *redundant* if $F = \lfloor F \rfloor$ in the above inference rule.

`weak_rhs`: Given $F \leq F'$ and a linear inequality $EX \leq F$ we can derive $EX \leq F'$

$$\frac{EX \leq F}{EX \leq F'} \qquad F \leq F'$$

We say an application of the `weak_rhs` rule is *redundant* if $F = F'$ in the above inference rule.

A *cutting plane proof* of an inequality $EX \leq F$ from $AX \leq B$ is a sequence of inequalities $E_1X \leq F_1, \ldots, E_lX \leq F_l$ such that

$$\frac{AX \leq B, E_1X \leq F_1, \ldots, E_{i-1}X \leq F_{i-1}}{E_iX \leq F_i} \qquad \texttt{nonneg\_lin\_comb or rounding}$$

for each $i = 1, \ldots, l$ and each step is an application of the `nonneg_lin_comb` or the `rounding` inference rules ($E_1, \ldots, E_l$ are rational row vectors and $F_1, \ldots, F_l$ are rational numbers). We do not need the `weak_rhs` rule anywhere, except possibly as the last step in a cutting plane proof.

$$\frac{E_lX \leq F_l}{EX \leq F} \qquad E = E_l, F_l \leq F'.$$

The cutting plane proof system provides a sound and complete inference system for integer linear inequalities. This is stated formally in the following theorem.

Theorem 16 *(Schrijver [129]) We are given a system of integer linear inequalities $AX \leq B$, where A, B are rational matrices and X is a column vector of integer variables. Let $EX \leq F$ be an inequality, where E is a rational row vector and F is a rational number.*
1. $AX \leq B$ has an integral solution and $AX \leq B$ implies $EX \leq F$ if and only if there is a cutting plane proof of $EX \leq F$ from $AX \leq B$.
2. $AX \leq B$ has no integral solution if and only if then there is a cutting plane proof of $0 \leq -1$ from $AX \leq B$.

We need to prove the following:

Lemma 6: *The following are equivalent:*

1. A system of LDEs $AX = B$ implies a LDE $EX = F$

2. $AX = B$ has no integral solution or there exists a rational row vector R such that $E = RA$ and $F = RB$.

Proof. (2) $\Rightarrow$ (1) is straightforward.

(1) $\Rightarrow$ (2): Given $AX = B$ implies a linear equation $EX = F$. If $AX = B$ has no integral solution we are done, that is, (2) holds. Otherwise, assume that $AX = B$ has an integral solution.

We can write $AX = B$ as an equivalent system of inequalities $AX \leq B \wedge -AX \leq -B$. The cutting plane (CP) proof rules provide a complete inference system for integer linear inequalities. We can write the LDE $EX = F$ as $EX \leq F \wedge -EX \leq -F$. The system of linear inequalities $AX \leq B \wedge -AX \leq -B$ implies $EX \leq F \wedge -EX \leq -F$. Let us consider the CP proof of $EX \leq F$ from the inequalities $AX \leq B \wedge -AX \leq -B$. We show that the inference rules used in this proof will only involve `nonneg_linear_comb` rule. Any application of `rounding` or `weak_rhs` rule will either be redundant or will lead to a contradiction. The later case is not possible because $AX = B$ or the equivalent system of inequalities has an integral solution.

Consider the first application of `rounding` in the CP proof of $EX \leq F$.

$$\frac{E_iX \leq F_i}{E_iX \leq \lfloor F_i \rfloor} \qquad E_i \in \mathbb{Z}^n$$

Since all the rules used to derive $E_iX \le F_i$ are non negative linear combination rules, we can combine all steps used to derive $E_iX \le F_i$ by a single application of the `nonneg_lin_comb` rule. That is, we can find rational row vector $[R_1, R_2]$ such that

$$\frac{\begin{bmatrix} A \\ -A \end{bmatrix} X \le \begin{bmatrix} B \\ -B \end{bmatrix}}{\underbrace{[R_1, R_2]\begin{bmatrix} A \\ -A \end{bmatrix} X}_{E_iX} \le \underbrace{[R_1, R_2]\begin{bmatrix} B \\ -B \end{bmatrix}}_{F_i}} \qquad [R_1, R_2] \ge 0$$

where R_1, R_2 are non-negative, $E_i = R_1A + R_2(-A)$ and $F_i = R_1B + R_2(-B)$. We can also derive $-E_iX \le -F_i$ by taking a non negative linear combination of $AX \le B \wedge -AX \le -B$ using $[R_2, R_1]$. If $F_i = \lfloor F_i \rfloor$ then the application of rounding rule

$$\frac{E_iX \le F_i}{E_iX \le \lfloor F_i \rfloor} \qquad E_i \in \mathbb{Z}^n$$

is redundant. Otherwise, let $\lfloor F_i \rfloor = k (\ne F_i)$ and

$$\frac{E_iX \le F_i}{E_iX \le k}$$

Since $\lfloor -F_i \rfloor = -k - 1$. We apply apply rounding to $-E_iX \le -F_i$ to obtain

$$\frac{-E_iX \le -F_i}{-E_iX \le -k-1} \qquad -E_i \in \mathbb{Z}^n$$

By combining the above two equations ($E_iX \leq k$ and $-E_iX \leq -k-1$) we obtain an equation $0 \leq -1$. But this means that the original system of inequalities $AX \leq B \wedge -AX \leq -B$ has no integral solution, which contradicts our assumption. Thus, the first application of the `rounding` rule in the CP proof must be redundant. Using similar reasoning (induction on the length of the proof) we can conclude that all applications of `rounding` in the CP proof must be redundant.

In the CP proof system described above there can be only one application of `weak_rhs` rule as the last step in a CP proof. We now show that the application of `weak_rhs` at the end of the CP proof must be redundant.

$$\frac{EX \leq F_l}{EX \leq F} \qquad\qquad F_l \leq F.$$

If $F_l = F$, then the application of `weak_rhs` is redundant. Otherwise, suppose $F_l < F$. Recall, that $-EX \leq -F$ is also an implied inequality of the original system. We can add $-EX \leq -F$ and $EX \leq F_l$ to obtain $0 \leq F_l - F$. Since $F_l < F$ we can divide $0 \leq F_l - F$ by positive rational number $F - F_l$, to obtain the equation $0 \leq -1$. But this is a contradiction.

Thus, the cutting plane proof of $EX \leq F$ can only involve redundant applications of `rounding` or `weak_rhs` rules. These applications of `rounding` or `weak_rhs` rules can be removed to obtain a derivation of $EX \leq F$ that only involves `nonneg_linear_comb` rule. All applications of `nonneg_linear_comb` rule in a CP proof can be combined to obtain a vector $[S_1, S_2]$ such that

$$\frac{\begin{bmatrix} A \\ -A \end{bmatrix} X \leq \begin{bmatrix} B \\ -B \end{bmatrix}}{\underbrace{[S_1, S_2] \begin{bmatrix} A \\ -A \end{bmatrix} X}_{EX} \leq \underbrace{[S_1, S_2] \begin{bmatrix} B \\ -B \end{bmatrix}}_{F}} \qquad [S_1, S_2] \geq 0$$

where S_1, S_2 are non-negative, $E = S_1 A + S_2(-A)$ and $F = S_1 B + S_2(-B)$. (Note that a proof of $-EX \leq -F$ can be obtained by taking a non negative linear combination of $AX \leq B, -AX \leq -B$ using $[S_2, S_1]$.) Thus, there exists a rational vector $R = S_1 - S_2$ such that $E = RA$ and $F = RB$. This shows (2) holds. $\square$

D.5 Proof of Lemma 7

We use the following result in the proof.

Theorem 17 *(Schrijver [129]) Let $AX = B$ be a system of LDEs, where A, B are rational matrices and X is a column vector of n integer variables. If $AX = B$ is satisfiable (has an integral solution), then we can find in polynomial time integral vectors $X_0, \ldots, X_t \in \mathbb{Z}^n$ such that*

$$\{X | AX = B; X \textit{ integral}\} = \{X_0 + \lambda_1 X_1 + \ldots + \lambda_t X_t | \lambda_1, \ldots, \lambda_t \in \mathbb{Z}\}$$

with $X_1,\ldots,X_t$ linearly independent. (We think of $X_0,X_1,\ldots,X_t \in \mathbb{Z}^n$ as column vectors.)

Example 44 Consider a system of LDEs $AX = B$:

$$\begin{bmatrix} 2 & 6 & 3 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$$

The set S of solutions to $AX = B$ is given as:

$$S = \left\{ \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} + \lambda_1 \begin{bmatrix} -3 \\ 3 \\ -4 \end{bmatrix} \middle| \lambda_1 \in \mathbb{Z} \right\} = \left\{ \begin{bmatrix} 2-3\lambda_1 \\ 3\lambda_1 \\ -4\lambda_1 \end{bmatrix} \middle| \lambda_1 \in \mathbb{Z} \right\}$$

Lemma 7: *Let $AX = B$ denote a system of LDEs, where A,B are rational matrices and X is a column vector of integer variables. Let $C_iX = D_i$ denote a LDE for $1 \leq i \leq m$ (C_i is a rational row vector and D_i is a rational number). The following are equivalent:*

1. $AX = B$ implies $\bigvee_{i=1}^{m} C_iX = D_i$

2. There exists a $1 \leq k \leq m$ such that $AX = B$ implies $C_kX = D_k$.

Proof. (2) $\Rightarrow$ (1): This direction of the proof is straightforward.

(1) $\Rightarrow$ (2): If $AX = B$ has no integral solution, then $AX = B$ implies any linear equation. Thus, (2) holds.

Assume that $AX = B$ has an integral solution. In this case we can use the theorem 17 and write the set S of all integral solutions to $AX = B$ as

$$S := \{X_0 + \lambda_1 X_1 + \ldots + \lambda_t X_t | \lambda_1, \ldots, \lambda_t \in \mathbb{Z}\}$$

where $X_0, X_1, \ldots, X_t \in \mathbb{Z}^n$ (assuming X has size $n \times 1$).

By substituting $X = X_0 + \lambda_1 X_1 + \ldots + \lambda_t X_t$ (with $\lambda_1, \ldots, \lambda_t$ as symbolic variables) in $C_i X - D_i$ we obtain

$$C_i(X_0 + \lambda_1 X_1 + \ldots + \lambda_t X_t) - D_i.$$

Since $C_i X_0, \ldots, C_i X_t$ are scalars (rational numbers), the difference $C_i X - D_i$ for $X \in S$ is a linear expression in $\lambda_1, \ldots, \lambda_t$. We denote the difference $C_i X - D_i$ for $X \in S$ by δ_i. It follows that

$$\left.\begin{array}{rcl}
\delta_1 & = & u_{10} + u_{11}\lambda_1 + \ldots + u_{1t}\lambda_t \\
 & \ldots & \\
\delta_i & = & u_{i0} + u_{i1}\lambda_1 + \ldots + u_{it}\lambda_t \\
 & \ldots & \\
\delta_m & = & u_{m0} + u_{m1}\lambda_1 + \ldots + u_{mt}\lambda_t
\end{array}\right\} EQ$$

where u_{ij} are rational numbers, $\lambda_1, \ldots, \lambda_t$, $\delta_1, \ldots, \delta_m$ are symbolic variables. An integral assignment $\lambda_1 = \beta_1, \ldots, \lambda_t = \beta_t$ where $\beta_1, \ldots, \beta_t \in \mathbb{Z}$ gives a solution $X_\beta \in \mathbb{Z}^n$ to $AX = B$ ($X_\beta \in S$). If δ_i evaluates to zero for $\lambda_1 = \beta_1, \ldots, \lambda_t = \beta_t$, then

X_β satisfies the LDE $C_iX = D_i$. Otherwise, X_β does not satisfy the LDE $C_iX = D_i$.

We consider two cases.

Case 1: If for some $1 \leq k \leq m$, $u_{k0} = \ldots = u_{kt} = 0$, then $\delta_k = 0$. That is, every $X \in S$ satisfies $C_kX = D_k$. Therefore, $AX = B$ implies $C_kX = D_k$. In this case (2) holds.

Case 2: For all $1 \leq k \leq m$ there is a $0 \leq j \leq t$ such that $u_{kj} \neq 0$. We show that case 2 cannot arise using proof by contradiction. We will give an algorithm for assigning integral values to $\lambda_1, \ldots, \lambda_t$ such that $\delta_1 \neq 0, \ldots, \delta_m \neq 0$. In other words, we will show that there exists an $X' \in S$ such that $C_iX' \neq D_i$ for all $1 \leq i \leq m$. This will mean that $AX = B$ does not imply $\vee_{i=1}^{m} C_iX = D_i$, leading to a contradiction.

It is convenient to think of expressions for $\delta_1, \ldots, \delta_m$ as a system of equations in $\delta_1, \ldots, \delta_m, \lambda_1, \ldots, \lambda_t$. We denote this system of equations as EQ.

We now give an algorithm for assigning integral values to $\lambda_1, \ldots, \lambda_t$ such that $\delta_1 \neq 0, \ldots, \delta_m \neq 0$. Our algorithm will assign λ_i before λ_{i+1} for each $1 \leq i \leq m-1$.

Let $EQ_0 \subseteq EQ$ denote the equations that do not contain any variables $\lambda_1, \ldots, \lambda_t$. If $\delta_k = u_{k0}$ is an equation in EQ_0, then we know that $u_{k0} \neq 0$ (by case 2 assumption). Thus, $C_kX \neq D_k$ for any $X \in S$. Alternatively, $AX = B$ cannot imply $C_kX = D_k$. We can safely ignore the equations in EQ_0 for the rest of the proof.

Let $EQ_i \subseteq EQ$ for $1 \leq i \leq t$ denote the set of equations which contain only variables $\lambda_1, \ldots, \lambda_i$ such that the coefficient of λ_i is not zero (coefficients of $\lambda_1, \ldots, \lambda_{i-1}$ can be zero).

We now describe an algorithm for assigning integer values to λ_i for $1 \leq i \leq t$. The algorithm uses EQ_i to assign a value to λ_i. Suppose we have assigned integral values $\alpha_1, \ldots, \alpha_{i-1}$ to $\lambda_1, \ldots, \lambda_{i-1}$, respectively. If $EQ_i = \emptyset$, then assign an arbitrary integer value α_i to λ_i. Otherwise, substitute $\lambda_1 = \alpha_1, \ldots, \lambda_{i-1} = \alpha_{i-1}$ in EQ_i to obtain a system of equations EQ_i'. A representative equation in EQ_i' is

$$\delta_l = v_{l0} + u_{li}\lambda_i \qquad\qquad u_{li} \neq 0$$

where v_{l0} is a rational number and u_{li} is a non-zero rational number by definition of EQ_i. We want to assign λ_i such that $\delta_l \neq 0$ for every equation $\delta_l = v_{l0} + u_{li}\lambda_i$ in EQ_i'. This can be done by assigning λ_i any integer value that is different from $\frac{-v_{l0}}{u_{li}}$. Let

$$\lambda_i := \alpha_i \qquad \text{where} \qquad \alpha_i \in \mathbb{Z} \qquad \text{and} \qquad \alpha_i \notin \left\{ \frac{-v_{l0}}{u_{li}} | l \in EQ_i' \right\}$$

where $l \in EQ_i'$ is a short form of saying that equation $\delta_l = v_{l0} + u_{li}\lambda_i$ is in EQ_i'. We can always find a suitable α_i because the set of integers has infinite cardinality (and we have a finite set of rational numbers/integers that cannot be assigned to λ_i).

Let $\delta_l = u_{l0} + \sum_{j=1}^{i} u_{lj}\lambda_j$ denote an equation in $EQ_1 \cup \ldots \cup EQ_i$. The following invariant holds after λ_i is assigned α_i: if $\lambda_1 = \alpha_1, \ldots, \lambda_i = \alpha_i$ is substituted in $\delta_l = u_{l0} + \sum_{j=1}^{i} u_{lj}\lambda_j$, then $\delta_l \neq 0$.

Thus, once we have assigned $\lambda_1 = \alpha_1, \ldots, \lambda_t = \alpha_t$ using the above algorithm we have $\delta_1 \neq 0, \ldots, \delta_m \neq 0$. Let $X' \in S$ be an integral solution to $AX = B$ given by $\lambda_1 = \alpha_1, \ldots, \lambda_t = \alpha_t$. Then $\delta_i = C_i X' - D_i \neq 0$ for each $1 \leq i \leq m$. That is, $AX = B$ does not imply $\vee_{i=1}^{m} C_i X = D_i$, leading to a contradiction. Thus, Case 2 cannot arise. □

D.6 Proof of Theorem 15

In addition to lemmas 6,7 we will use the following theorem.

Theorem 18 *(Schrijver [129]) Let A be a rational matrix, B be a rational column vector, C be a rational row vector. Assume that the system $AX = B$ has a rational solution. Then $AX = B$ implies (over rationals) $CX = D$ if and only if there is a row vector R such that $RA = C$ and $RB = D$.*

Theorem 15. *Let F denote $AX = B \wedge \bigwedge_{i=1}^{m} C_i X \neq D_i$. The following are equivalent:*

1. F has no integral solution

2. F has no rational solution or $AX = B$ has no integral solution.

Proof. (2) $\Rightarrow$ (1) is straightforward.

(1) $\Rightarrow$ (2): Given F has no integral solution. If $AX = B$ has no integral solution, then (2) holds. Otherwise, assume $AX = B$ has an integral solution. Since F has no integral solution, every integral solution to $AX = B$ must satisfy $C_i X = D_i$ for

some $1 \le i \le m$. That is,

$$AX = B \Rightarrow \bigvee_{i=1}^{m} C_i X = D_i$$

By lemma 7 it follows that there exists a $1 \le k \le m$ such that

$$AX = B \Rightarrow C_k X = D_k$$

By lemma 6 (and our assumption that $AX = B$ has an integral solution) it follows that there exists a rational row vector R such that

$$C_k = RA \qquad \text{and} \qquad D_k = RB$$

Using the vector R and theorem 18 we can conclude that $AX = B$ implies $C_k X = D_k$ over rationals. So

$$AX = B \wedge C_k X \neq D_k$$

is unsatisfiable over rationals, or

$$AX = B \wedge \bigwedge_{i=1}^{m} C_i X \neq D_i$$

is unsatisfiable over rationals. Thus, F is unsatisfiable over rationals and (2) holds.
$\square$

D.7 Interpolants for Linear Diophantine Equations and Disequations (LDEs+LDDs)

We use the following theorem.

Theorem 19 *(Schrijver [129]) Let A be a rational matrix, B be a rational column vector. The system $AX = B$ has no rational solution if and only if there exists a rational row vector R such that $RA = 0$ and $RB \neq 0$.*

Let $F \wedge G$ be systems of LDEs+LDDs.

$$\begin{aligned} F &:= AX = B \wedge \bigwedge_i C_i X \neq D_i \\ G &:= A'X = B' \wedge \bigwedge_j C'_j X \neq D'_j \end{aligned}$$

$F \wedge G$ represents another system of LDEs+LDDs. Suppose $F \wedge G$ is unsatisfiable (no integral solution). In this case we want to compute an interpolant for the pair (F, G). We divided this problem into two cases in Section 6.7. We describe Case 1 below.

By case 1 assumption we know that $F \wedge G$ has no rational solution. We want to compute an interpolant for (F, G). The interpolant for (F, G) can be obtained by using the techniques discussed in [113, 144, 128, 54]. For completeness we show how to obtain an interpolant for (F, G) by considering three sub-cases.

Case 1.1: $AX = B \wedge A'X = B'$ has no rational solution. Using theorem 19 there exists a row vector $[R_1, R_2]$ such that

$$R_1A + R_2A' = 0$$
$$R_1B + R_2B' \neq 0$$

In this case an interpolant for the pair (F, G) is the linear equation $R_1AX = R_1B$. One can verify that $R_1AX = R_1B$ satisfies all the conditions required by the definition of interpolants.

We describe Case 1.2 and Case 1.3 next. Since $F \wedge G$ is unsatisfiable over rationals we have

$$AX = B \wedge A'X = B' \Rightarrow (\bigvee_i C_iX = D_i \vee \bigvee_j C'_jX = D'_j) \tag{D.15}$$

The above implication holds for any rational X. We know that if a set of rational linear arithmetic constraints Γ imply a disjunction of linear equations $\bigvee_{i=1}^{m} Eq_i$, then for some $1 \leq k \leq m$, Γ implies Eq_k. This is due to *convexity* of rational linear arithmetic [120].

Due to convexity $AX = B \wedge A'X = B'$ will imply either an equality belonging to $\bigvee_i C_iX = D_i$ or an equality belonging to $\bigvee_j C'_jX = D'_j$ in equation D.15. This gives Case 1.2 and Case 1.3.

Case 1.2: For some j, $AX = B \wedge A'X = B' \Rightarrow C'_j X = D'_j$.

Using theorem 18 there exists a row vector $[R_1, R_2]$ such that

$$\begin{aligned} R_1A + R_2A' &= C'_j \\ R_1B + R_2B' &= D'_j. \end{aligned}$$

In this case an interpolant for (F, G) is the linear equation $R_1AX = R_1B$. One can verify that $R_1AX = R_1B$ satisfies all the conditions required by the definition of interpolants.

Case 1.3: For some i, $AX = B \wedge A'X = B' \Rightarrow C_iX = D_i$.

In the above two cases (1.1 and 1.2) the interpolant is a linear equation. In this case the interpolant will be a linear disequation. Using theorem 18 there exists a row vector $[R_1, R_2]$ such that

$$R_1A + R_2A' = C_i$$
$$R_1B + R_2B' = D_i$$

Let V_{FG} denote the variables that occur in both F and G and let $V_{F \setminus G}$ denote the variables that occur only in F (and not in G).

Observe that $R_1AX = R_1B$ can be written as follows:

$$\sum_{x_i \in V_{F \setminus G}} a_i x_i + \sum_{x_i \in V_{FG}} b_i x_i = k$$

Similarly, $C_iX = D_i$ can be written as follows:

$$\sum_{x_i \in V_{F \setminus G}} a_i x_i + \sum_{x_i \in V_{FG}} c_i x_i = D_i$$

Observe that the variables $x_i \in V_{F \setminus G}$ have same coefficients in R_1AX and C_iX. This is because $C_i = R_1A + R_2A'$ and the coefficients of $x_i \in V_{F \setminus G}$ in $R_2A'X$ is zero.

We can write $C_iX \neq D_i$ as

$$\sum_{x_i \in V_{F \setminus G}} a_i x_i + \sum_{x_i \in V_{FG}} c_i x_i \neq D_i$$

Note that F implies $R_1AX = R_1B$ and $C_iX \neq D_i$. Thus, F implies the disequation obtained by subtracting $R_1AX = R_1B$ and $C_iX \neq D_i$.

$$\sum_{x_i \in V_{FG}} b_i x_i - \sum_{x_i \in V_{FG}} c_i x_i \neq k - D_i$$

The above equation is the required interpolant. It it implied by F and only contains variables common to F, G. One can show that above disequation is $R_2A'X \neq R_2B'$. Since G implies $R_2A'X = R_2B'$ the above equation is unsatisfiable with G.

D.8 Handling of Linear Modular Disequations

Lemma 13 *The problem of deciding whether a system (conjunction) of linear modular disequations (LMDs) have an integral solution is NP-hard.*

Proof. We reduce a well known NP-hard problem 3-SAT to a system of LMDs denoted by $\mathcal{L}$. Let the variables in 3-SAT problem be $z_1, \ldots, z_n$. For each variable z_i in the 3-SAT problem we introduce two integer variables x_i and x'_i in $\mathcal{L}$, where x_i represents the literal z_i and x'_i represents the literal $\bar{z}_i$.

The modulus of LMDs in $\mathcal{L}$ will be four. We first express the constraints that $x_i \equiv_4 1$ and $x'_i \equiv_4 0$ or $x_i \equiv_4 0$ and $x'_i \equiv_4 1$. This done by means of the following LMDs.

$$\begin{aligned}\mathcal{L}_1 \;:=\; & \bigwedge_{i=1}^{n} \neg(x_i \equiv_4 x'_i) \quad \wedge \quad \bigwedge_{i=1}^{n} \neg(x_i \equiv_4 2) \wedge \bigwedge_{i=1}^{n} \neg(x_i \equiv_4 3) \wedge \\ & \bigwedge_{i=1}^{n} \neg(x'_i \equiv_4 2) \wedge \bigwedge_{i=1}^{n} \neg(x'_i \equiv_4 3)\end{aligned}$$

Now consider any clause $u \vee v \vee w$ in the given 3-SAT formula, where $u, v, w \in \{z_1, \ldots, z_n, \bar{z_1}, \ldots, \bar{z_n}\}$. Let $\delta(u)$ map the literal u to the corresponding variable in $\mathcal{L}$. For each clause $u \vee v \vee w$ in the 3-SAT formula, we generate the following LMD

$$\neg(\delta(u) + \delta(v) + \delta(w) \equiv_4 0).$$

The LMD above is falsified only when $\delta(u), \delta(v), \delta(w)$ are assigned 0 $(\mathit{mod}\ 4)$. For all other assignment of values $\delta(u), \delta(v), \delta(w)$ the LMD is satisfied (captures the semantics of the clause).

Let the set of clauses in the 3-SAT formula be C.

$$\mathcal{L}_2 \;:=\; \bigwedge_{(u \vee v \vee w) \in C} \neg(\delta(u) + \delta(v) + \delta(w) \equiv_4 0)$$

Let $\mathcal{L} = \mathcal{L}_1 \wedge \mathcal{L}_2$. Observe that the 3-SAT formula is satisfiable if and only if $\mathcal{L}$ is satisfiable. The reduction from the given 3-SAT formula to $\mathcal{L}$ is polynomial time. This establishes the NP-hardness of checking the satisfiability of conjunctions of LMDs. $\square$

D.8.1 Proofs of Unsatisfiability and Interpolants for LMDs

We can reduce a system of LMDs or LMEs+LMDs to a conjunction of atomic formulas in integer linear arithmetic (both problems are NP-hard) and use the *cutting-plane proof system* to obtain a proof of unsatisfiability. Pudlak's [126] algorithm can be used for obtaining interpolants.

D.9 Obtaining Polynomially Sized Cutting-plane Proofs for LDEs

Given an unsatisfiable system of LDEs $AX = B$, a proof of unsatisfiability is a rational row vector R such that RA is integral, while RB is not an integer. We know that R can be obtained in polynomial time.

We show that using R we can obtain a polynomially sized cutting plane proof of unsatisfiability of $AX = B$. The cutting plane proof system was described in Appendix D.4. It consists of three inference rules `nonneg_lin_comb`, `rounding` and `weak_rhs`.

We first write $R = S_1 - S_2$, where both S_1, S_2 are non-negative row vectors. For example, we can write $[\frac{1}{2}, -\frac{3}{4}] = [\frac{1}{2}, 0] - [0, \frac{3}{4}]$.

We write $AX = B$ as $AX \leq B \wedge -AX \leq -B$. The cutting plane proof of unsatisfiability consists of following steps.

$$\frac{AX \leq B}{S_1AX \leq S_1B} \qquad S_1 \geq 0 \qquad \texttt{nonneg\_lin\_comb}$$

$$\frac{-AX \leq -B}{-S_2AX \leq -S_2B} \qquad S_2 \geq 0 \qquad \texttt{nonneg\_lin\_comb}$$

$$\frac{S_1AX \leq S_1B \qquad -S_2AX \leq -S_2B}{[S_1 - S_2]AX \leq [S_1 - S_2]B} \qquad \texttt{nonneg\_lin\_comb}$$

Since $R = [S_1 - S_2]$ we can write the above step as

$$\frac{S_1AX \leq S_1B \qquad -S_2AX \leq -S_2B}{RAX \leq RB} \qquad \texttt{nonneg\_lin\_comb}$$

Multiplying $AX \leq B$ by S_2 and $-AX \leq -B$ by S_1 we can derive

$$\frac{S_2AX \leq S_2B \qquad -S_1AX \leq -S_1B}{-RAX \leq -RB} \qquad \texttt{nonneg\_lin\_comb}$$

By definition of R we know that RB is not an integer. Let $\lfloor RB \rfloor = k$. Then $\lfloor -RB \rfloor = -k-1$. Since RA is integral we can apply rounding to $RAX \leq RB$ and $-RAX \leq -RB$.

$$\frac{RAX \leq RB}{RAX \leq k} \qquad \texttt{rounding}$$

$$\frac{-RAX \leq -RB}{RAX \leq -k-1} \qquad \texttt{rounding}$$

The contradiction is obtained by summing $RAX \leq k$ and $RAX \leq -k-1$.

$$\frac{RAX \leq RB \qquad -RAX \leq -RB}{0 \leq -1} \qquad \texttt{nonneg\_lin\_comb}$$

Since R is polynomially sized the cutting plane proof is also polynomially sized.

D.10 Using SMT Solvers for Obtaining a Proof of Unsatisfiability for LDEs/LMEs

We can determine if a system of LDEs $CX = D$ is unsatisfiable and obtain a proof of unsatisfiability (if applicable) by using decision procedures for (mixed) integer linear arithmetic in a black-box fashion. For example, one can use modern SMT solvers such as Yices [24] to obtain proofs of unsatisfiability. The idea is to encode the existence of a rational row vector R such that RC is integral and RD is not an integer in form of a formula that can be checked using existing decision procedures. This is motivated by the idea proposed in [128] for real and rational linear arithmetic. We illustrate the technique by means of an example.

Example 45 Consider the system of LDEs $CX = D$:

$$\begin{bmatrix} 1 & -2 & 0 \\ 1 & 0 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

We use two rational variables r_1, r_2 to denote the proof of unsatisfiability $R = [r_1, r_2]$. We use three integer variables v_1, v_2, v_3 to express the constraint that RC is integral. We introduce another integer variable v_4 to express the constraint that $RD = r_2$ is not an integer.

$$P := (v_1 = r_1 + r_2) \wedge (v_2 = -2r_1) \wedge (v_3 = -2r_2) \wedge (v_4 < r_2) \wedge (r_2 < v_4 + 1)$$

If the decision procedure for integer linear arithmetic determines that P is satisfiable, then we get a proof of unsatisfiability for $CX = D$ by looking at the assignments to r_1, r_2. If P is unsatisfiable, it means that the system $CX = D$ is satisfiable.

We formalize the idea below. Suppose the sizes of C, X, D in the system of LDEs $CX = D$ are $m \times n, n \times 1, m \times 1$, respectively. The formula P contains:

- m rational variables $r_1, \ldots, r_m$ such that $R = [r_1, \ldots, r_m]$
- n integer variables $v_1, \ldots, v_n$ to express that each element of RC is integral.
- One integer variable v_{n+1} to express the constraint RD is not an integer by using two strict inequalities

Let $(RC)_i$ denote the *ith* element in the row vector RC. Then we have

$$P := \bigwedge_{i=1}^{n} v_i = (RC)_i \ \wedge\ (v_{n+1} < RD)\ \wedge\ (RD < v_{n+1} + 1)$$

The formula P is given to a SMT solver. If P is satisfiable, we get the required proof of unsatisfiability R. Otherwise, we know that the given system of LDEs is satisfiable.

The proof of unsatisfiability for a system of linear modular equations can be computed in a similar manner as well (using definition 17).

As shown by experimental results in Section 6.8, the black-box use of SMT solver Yices to obtain proofs of unsatisfiability is not efficient (as compared to the use of HNF). The main reason for this seems to be the structure of P. Even though the encoding used to obtain P is natural, it is difficult for algorithms used in Yices to decide P.