SCALABLE QUANTIFIER ELIMINATION TECHNIQUES FOR FORMAL VERIFICATION

By

Ajith K J

ENGG01200804016

Bhabha Atomic Research Centre, Mumbai

A thesis submitted to the

Board of Studies in Engineering Sciences

In partial fulfillment of requirements

for the degree of

DOCTOR OF PHILOSOPHY

of

HOMI BHABHA NATIONAL INSTITUTE

April, 2017

Homi Bhabha National Institute

Recommendations of the Viva Voce Committee

As members of the Viva Voce Committee, we certify that we have read the dissertation prepared by Ajith K J entitled “Scalable Quantifier Elimination Techniques for Formal Verification” and recommend that it may be accepted as fulfilling the thesis requirement for the award of Degree of Doctor of Philosophy.

Chairperson - Prof. (Mrs.) Archana Sharma	Date:
Guide - Prof. Supratik Chakraborty, IIT Bombay	Date:
Examiner - Prof. Pallab Dasgupta, IIT Kharagpur	Date:
Member 1 - Prof. V. H. Patankar	Date:
Member 2 - Prof. (Mrs.) Gopika Vinod	Date:
Technology Adviser - Prof. A. K. Bhattacharjee	Date:

Final approval and acceptance of this thesis is contingent upon the candidate’s submission of the final copies of the thesis to HBNI.

I hereby certify that I have read this thesis prepared under my direction and recommend that it may be accepted as fulfilling the thesis requirement.

Date:

Place: **Guide**

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfilment of requirements for an advanced degree at Homi Bhabha National Institute (HBNI) and is deposited in the Library to be made available to borrowers under rules of the HBNI.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the Competent Authority of HBNI when in his or her judgement the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

Ajith K J

DECLARATION

I, hereby declare that the investigation presented in the thesis has been carried out by me. The work is original and has not been submitted earlier as a whole or in part for a degree/diploma at this or any other Institution/University.

Ajith K J

List of publications arising from the thesis

Journal

1. Ajith K. J., S. Chakraborty. *A Layered Algorithm for Quantifier Elimination from Linear Modular Constraints*, Formal Methods in System Design, Volume 49, Issue 3, December 2016, Pages 272-323.

Conferences

1. Ajith K. J., S. Chakraborty. *A Quantifier Elimination Algorithm for Linear Modular Equations and Disequations*, International Conference on Computer Aided Verification (CAV) 2011, USA, 14-20 July 2011 (appears in Lecture Notes in Computer Science, Volume 6806, 2011, Pages 486-503).

2. Ajith K. J., S. Chakraborty. *Extending Quantifier Elimination to Linear Inequalities on Bit-vectors*, International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2013, Italy, 16-24 March 2013 (appears in Lecture Notes in Computer Science, Volume 7795, 2013, Pages 78-92).

3. Ajith K. J., S. Chakraborty. *Quantifier Elimination for Linear Modular Constraints*, International Congress on Mathematical Software (ICMS) 2014, South Korea, 5-9 August 2014 (appears in Lecture Notes in Computer Science, Volume 8592, 2014, Pages 295-302).

4. Ajith K. J., S. Shah, S. Chakraborty, A. Trivedi, S. Akshay. *A CEGAR Algorithm for Generating Skolem Functions for Factored Formulas*, Inter-

national Conference on Formal Methods in Computer-Aided Design (FM-CAD) 2015, USA, 27-30 September 2015 (appears in Proceedings of FM-CAD 2015, Pages 85-92).

Others

1. Ajith K. J., A. K. Bhattacharjee, M. Sharma, G. Ganesh, S. D. Dhodapkar, B. Biswas. *Design and Application of a Formal Verification Tool for VHDL Designs*, In BARC Newsletter, ISSN: 0976-2108, March-April 2012, http://www.barc.gov.in/publications/nl/2012/2012030412.pdf.

2. Ajith K. J., A. K. Bhattacharjee. *VBMC: A Formal Verification Tool for VHDL Programs*, BARC External Technical Report, BARC/2014/E/010, http://www.iaea.org/inis/collection/NCLCollectionStore/_Public/45/102/45102099.pdf.

Dedicated to my wife and children

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my guide Prof. Supratik Chakraborty for his invaluable guidance, suggestions and encouragement during the course of this work.

I am extremely thankful to Dr. A. K. Bhattacharjee and S. D. Dhodapkar for their indispensable help and support throughout this work. I am indebted to Dr. Shetal Shah, Prof. S. Akshay and Prof. Ashutosh Trivedi for the discussions and suggestions during the work.

I am grateful to my doctoral committee members Dr. Kallol Roy, Dr. Archana Shrama, Dr. Gopika Vinod, and Dr. V. H. Patankar for their valuable suggestions. My sincere thanks to Dr. Gopinath Karmakar whose suggestions helped me a lot during the course of this work.

I thank my colleagues and friends Amol, Prateek, Ganesh, Hrishi and Abhishekh for their help and support. Finally, I thank my wife Nicy, children Robin and Neha, my parents and my brother for their love and encouragement.

Ajith K J

Synopsis

Quantifier elimination involves converting a logic formula containing quantifiers into a semantically equivalent quantifier-free formula. This has a number of important applications in formal verification and analysis of hardware and software systems. Many key operations performed by formal verification and analysis tools essentially boil down to quantifier elimination from logic formulas. This thesis presents practically efficient and scalable techniques for quantifier elimination from logic formulas that can improve the performance of such formal verification and analysis tools.

Boolean combinations of linear equalities, disequalities and inequalities on fixed-width bit-vectors, collectively called linear modular constraints, is an important fragment of the theory of fixed-width bit-vector logic. Quantifier elimination from linear modular constraints is extremely important in the context of formal verification and analysis of word-level RTL designs and embedded programs. The most dominant technique used for eliminating quantifiers from these constraints is bit-blasting, followed by bit-level quantifier elimination. Since linear modular constraints can be expressed as formulae in linear integer arithmetic, quantifier elimination for linear integer arithmetic can also be used. However, both the above approaches destroy the word-level structure of the problem, and

do not scale well for linear modular constraints with large bit-widths.

We present a practically efficient and bit-precise algorithm for quantifier elimination from conjunctions of linear modular constraints. Our algorithm uses a layered approach, whereby sound but incomplete and cheaper layers are invoked first, and expensive but complete layers are called only when required. Unlike alternative quantifier elimination techniques based on bit-blasting and conversion to linear integer arithmetic, our algorithm keeps the quantifier-eliminated formulae in linear modular arithmetic. We also extend this algorithm to work with arbitrary Boolean combinations of linear modular constraints. Experiments indicate that our techniques significantly outperform alternative quantifier elimination techniques. The experiments also demonstrate the utility of our techniques in bounded model checking of word-level RTL designs.

We then present Skolem function-based techniques for quantifier elimination from formulas in propositional logic. Techniques for generating Skolem functions are of significant interest not only in quantifier elimination, but also in certification of solvers, synthesis of programs and circuits from specifications, and disjunctive decomposition of sequential circuits. In many such applications, the input formula is given as a conjunction of simpler sub-formulas, called factors, each of which depends on a small subset of variables. Existing algorithms for Skolem function generation ignore any such factored form and treat the input formula as a monolithic conjunction of factors.

We present a SAT solving based algorithm for generating Skolem functions for propositional formulas that exploits factored representation of input formulas. In contrast to existing algorithms, our algorithm neither requires a proof of satisfiability nor uses composition of monolithic conjunctions of factors. Experiments

indicate that on several large problem instances, our algorithm generates smaller Skolem functions and runs faster when compared to existing Skolem function generation algorithms.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Quantifier elimination (QE) is the process of converting a logic formula containing quantifiers into a semantically equivalent quantifier-free formula. This has a number of important applications in formal verification and analysis of hardware and software systems. Key operations such as image computation [1], computation of strongest post-conditions [2] and computation of predicate abstractions [3] performed by formal verification and analysis tools essentially reduce to QE from logic formulas. The motivation for this dissertation is the development of practically efficient and scalable techniques for QE from logic formulas that can improve the performance of such formal verification and analysis tools.

In this thesis, we initially focus on QE from formulas in an important fragment of bit-vector logic [13] called linear modular arithmetic. Formulas in linear modular arithmetic are Boolean combinations of linear equalities, disequalities and inequalities on fixed-width bit-vectors. QE techniques for linear modular arithmetic have been found useful in formal verification and analysis of word-level RTL designs and embedded programs [80, 77]. The problem of QE from for-

mulas in linear modular arithmetic and the details of our work in this field are explained in Section 1.2.

We then address the problem of QE from formulas in propositional logic. One of the approaches for performing QE is to express the quantified variables as functions of other variables in the formula. Such functions are traditionally called Skolem functions [5]. QE can be done by substituting the occurrences of the quantified variables in the formula by their Skolem functions. We focus on techniques to generate Skolem functions for quantified variables in propositional logic formulas. Other than QE, the techniques for generating Skolem functions have important applications in reachability analysis of circuits [6, 7], certification of solvers [8] and synthesis of programs from specifications [9]. The details of our work in this field are explained in Section 1.3.

1.1 Preliminaries

Before delving into the details of our work, we provide a brief introduction to first-order logic, first-order theories and QE. More detailed discussion on these topics can be found in books on logic (see [5, 30]). We also introduce notation that will be used in the remainder of this thesis.

1.1.1 First-Order Logic

The alphabet of first-order logic consists of variables, logical symbols, and non-logical symbols. Logical symbols include Boolean connectives ($\neg$, $\vee$, $\wedge$), quantifiers ($\forall$, $\exists$), constant formulas (false, true), and parentheses. Nonlogical symbols include function symbols, predicate symbols, and constant symbols.

Terms and formulas in first-order logic are constructed using this alphabet. Terms are variables, constants, and function applications over other terms. false and true are called constant formulas. The most basic formulas are constructed by application of predicates over terms, and are called *atomic constraints* or simply *constraints*. More complex formulas are constructed by application of Boolean connectives and quantifiers over other formulas: (i) if F is a formula, then $\neg F$ is a formula, (ii) if F_1 and F_2 are formulas, then $F_1 \wedge F_2$ and $F_1 \vee F_2$ are formulas, (iii) if F is a formula and x is a variable appearing in F, then $\exists x.F$ and $\forall x.F$ are formulas.

A formula constructed only by using constraints and Boolean connectives is called a *quantifier-free* formula. The variables appearing in a quantifier-free formula are called *free* variables. Let F be a quantifier-free formula over a set V of free variables. We often write F as $F(V)$ to highlight the fact that F is a formula over variables in V. A formula of the form $Qx.F$, where $x \in V$ and $Q \in \{\exists, \forall\}$ is called a *quantified* formula. The variable x here is called *bound* variable in $Qx.F$, and the variables in $V \setminus \{x\}$ are called *free* variables $Qx.F$.

1.1.2 First-Order Theories

A first-order theory defines the set of nonlogical symbols that can be used in terms and formulas. It also defines the domain D of variables, and gives interpretation to the nonlogical symbols. The interpretation to the nonlogical symbols is a mapping from function symbols, predicate symbols, and constant symbols to functions, predicates, and elements over D.

As an example, Presburger arithmetic [40, 13] is a first-order theory with nonlogical symbols 0, 1, $+$ and $\leq$. The domain of variables in this theory is the set $\mathcal{Z}$

of integers. The symbols 0 and 1 map to the integers 0 and 1. The symbols $+$ and $\leq$ have their obvious interpretations over $\mathcal{Z}$. Linear real arithmetic, linear modular arithmetic, propositional logic etc. are other examples of first-order theories.

Given a formula F in a first-order theory $\mathcal{T}$, a *variable assignment* π assigns an element of the domain D to each free variable in F. The formula F evaluates to either true or false under π. If F evaluates to true under π, then we say $\pi \models_{\mathcal{T}} F$. Otherwise, if F evaluates to false under π, then we say $\pi \not\models_{\mathcal{T}} F$. When the theory $\mathcal{T}$ is clear from the context, we simply say $\pi \models F$ and $\pi \not\models F$. Note that each constraint c in F evaluates to either true or false under π depending on the interpretations of the nonlogical symbols in $\mathcal{T}$. Formulas with Boolean connectives are evaluated in the following manner: (i) $\pi \models_{\mathcal{T}} \neg F_1$ iff $\pi \not\models_{\mathcal{T}} F_1$, (ii) $\pi \models_{\mathcal{T}} F_1 \wedge F_2$ iff $\pi \models_{\mathcal{T}} F_1$ and $\pi \models_{\mathcal{T}} F_2$, (iii) $\pi \models_{\mathcal{T}} F_1 \vee F_2$ iff $\pi \models_{\mathcal{T}} F_1$ or $\pi \models_{\mathcal{T}} F_2$. In order to explain how quantified formulas are evaluated, we define the notation $\pi[x|d]$ for variable x and $d \in D$. We define $\pi[x|d]$ as a variable assignment which is exactly the same as π except that the variable x is assigned the value d. We have $\pi \models_{\mathcal{T}} \exists x.F_1$ iff $\pi[x|d] \models_{\mathcal{T}} F_1$ for some $d \in D$. Similarly, we have $\pi \models_{\mathcal{T}} \forall x.F_1$ iff $\pi[x|d] \models_{\mathcal{T}} F_1$ for all $d \in D$.

Let F, F_1, F_2 be formulas in a first-order theory $\mathcal{T}$. Formula F is called $\mathcal{T}$-*satisfiable* if there exists a variable assignment π such that $\pi \models_{\mathcal{T}} F$. Formula F is $\mathcal{T}$-*valid* if for every variable assignment π, we have $\pi \models_{\mathcal{T}} F$. Formulas F_1 and F_2 are $\mathcal{T}$-*equivalent* if for every variable assignment π, $\pi \models_{\mathcal{T}} F_1$ iff $\pi \models_{\mathcal{T}} F_2$. When the theory $\mathcal{T}$ is clear from the context, we simply use *satisfiable*, *valid*, and *equivalent*; we use $F_1 \equiv F_2$ to denote that the formulas F_1 and F_2 are equivalent. Moreover, we often use $F_1 \Rightarrow F_2$ to denote the formula $\neg F_1 \vee F_2$.

1.1.3 Quantifier Elimination

Let F be a quantifier-free formula over a set V of free variables in a first-order theory $\mathcal{T}$. Consider the quantified formula $Q_1x_1\,Q_2x_2\,\ldots\,Q_nx_n.F$, where $X = \{x_1,\ldots x_n\}$ is a subset of V, and $Q_i \in \{\exists,\forall\}$ for $i \in \{1,\ldots n\}$. QE involves computing a quantifier-free formula F' on variables in $V \setminus X$ such that F' is $\mathcal{T}$-equivalent to F.

As an example, consider the formula $\exists x.\,(y = 3x)$ in the theory of linear arithmetic over integers. This formula expresses the set y of integers that are divisible by 3. Since the quantifier-free formula $(y = 0 \pmod 3)$ expresses the set of integers that are divisible by 3, $\exists x.\,(y = 3x)$ is equivalent to $(y = 0 \pmod 3)$ in the theory of linear arithmetic over integers.

Given a universally quantified formula $\forall x.F$ in any first-order theory $\mathcal{T}$, $\forall x$ can be converted to $\exists x$ using the equivalence $\forall x.F \equiv \neg\exists x.\neg F$. Hence the problem of developing a QE algorithm for theory $\mathcal{T}$ boils down to the problem of developing an *existential* QE algorithm for $\mathcal{T}$. Therefore, in the remaining part of this thesis, we will use QE to refer to existential QE.

1.2 QE for Linear Modular Arithmetic

Linear modular arithmetic is a fragment of bit-vector logic [13]. Constraints in linear modular arithmetic are linear equalities, disequalities and inequalities on fixed-width bit-vectors. Let p be a positive integer constant, $x_1,\ldots,x_n$ be p-bit non-negative integer variables, and $a_0,\ldots,a_n$ be integer constants in $\{0,\ldots,2^p - 1\}$. A linear term over $x_1,\ldots,x_n$ is a term of the form $a_1\cdot x_1 + \cdots a_n\cdot x_n + a_0$, where $\cdot$ denotes multiplication modulo 2^p and $+$ denotes addition modulo 2^p. A linear

modular equality (LME) is a constraint of the form $t_1 = t_2 \pmod{2^p}$, where t_1 and t_2 are linear terms over $x_1, \ldots, x_n$. Similarly, a linear modular disequality (LMD) is a constraint of the form $t_1 \neq t_2 \pmod{2^p}$, and a linear modular inequality (LMI) is a constraint of the form $t_1 \bowtie t_2 \pmod{2^p}$, where $\bowtie \in \{<, \leq\}$. We will use linear modular constraint (LMC) to refer to an LME, LMD or LMI. Conventionally 2^p is called the modulus of the LMC.

The semantics of LMCs differ from that of linear constraints over integers in two aspects:

1. *Wrap-around behaviour:* The successor of $2^p - 1$ in modular arithmetic is 0. Hence, for $x = 2^p - 1$, the value of $x + 1$ modulo 2^p overflows and wraps to 0. Hence the formula $(x = 3) \wedge (x + 1 \leq 2)$ is satisfiable in linear modular arithmetic with modulus 4, whereas it is unsatisfiable over integers.

2. *Finite domain:* Domain of variables in modular arithmetic has finite/bounded cardinality unlike integer arithmetic where the variables are unbounded. Hence the formula $(x = 3) \wedge (x < y)$ is unsatisfiable in linear modular arithmetic with modulus 4, whereas it is satisfiable over integers.

Efficient techniques for QE from LMCs have applications in formal verification and analysis of hardware and software systems. Formal verification and analysis tools reason about symbolic transition relations of hardware and software systems expressed as formulas in appropriate logic. Symbolic transition relations of word-level RTL designs and embedded programs involve constraints in linear modular arithmetic. LMEs arise from the assignment statements, whereas LMDs and LMIs arise primarily from branch and loop conditions that compare words/registers. Key operations such as image computation [1], computation of

strongest post-conditions [2] and computation of predicate abstractions [3] performed by formal verification and analysis algorithms boil down to QE from formulas involving symbolic transition relation. Symbolic transition relations of RTL designs and embedded programs in general may involve signed variables with signed operations and comparisons on them. However, there are standard techniques to convert constraints with signed semantics into equisatisfiable constraints with unsigned semantics (for example, see page 2 of [81]). Therefore, we will focus only on constraints with unsigned semantics.

Our primary motivation for studying QE from LMCs arises from bounded model checking [4] of word-level RTL designs. Suppose we wish to use bounded model checking to prove that a property holds for the first N cycles of operation of a word-level RTL design. This can be done by unrolling the symbolic transition relation of the design N times, conjoining the unrolled relation with the negation of the property, and then checking for satisfiability of the resulting formula using an SMT solver [10]. Since the transition relation contains all variables that appear in the RTL description, unrolling the transition relation a large number of times gives a formula with a large number of variables. While the number of variables in a formula is not the only factor that affects the performance of an SMT solver, for large enough values of N, the increased variable count has an adverse effect on the performance of the solver.

In order to alleviate the above problem, one can use an abstract transition relation that relates only a chosen subset of variables relevant to the property being checked, while abstracting the relation between the other variables. Such an abstract transition relation can be computed by existentially quantifying out the variables that are not relevant to the property being checked from the original

transition relation. Since the abstract transition relation contains fewer variables than the original transition relation, the formula obtained by unrolling the abstract transition relation has fewer variables. In general, this can lead to significantly better performance of the SMT solver, as demonstrated in our experiments. Since Boolean combinations of LMCs arise in transition relations of word-level RTL designs, building an abstract transition relation requires existentially quantifying variables from Boolean combinations of LMCs.

For ease of analysis, formal verification and analysis algorithms abstract variables in the system to be verified as unbounded integers, and use QE techniques for unbounded integers [11]. However the underlying system implementation often uses modular arithmetic, and as mentioned earlier, the semantics of unbounded integer arithmetic differs from that of modular arithmetic. Hence the results of verification and analysis by abstracting variables as unbounded integers and using QE for unbounded integers may not be sound or complete [12] if the underlying implementation uses modular arithmetic. Therefore, developing *bit-precise* QE techniques for LMCs is an important problem.

Currently, the dominant technique for eliminating quantifiers from LMCs involves *blasting* variables into individual bits (also called bit-blasting [13]), followed by elimination of the blasted bit-level variables [14]. This approach has some undesirable features. Blasting involves a bitwidth-dependent blow-up in the size of the problem. This can present scaling problems in the usage of bit-level QE tools, especially when reasoning about wide words. Similarly, given an instance of the QE problem, blasting variables that are quantified may transitively require blasting other variables (that are not quantified) as well. This can cause the quantifier-eliminated formula to appear like a propositional formula on blasted

bits, instead of being a modular arithmetic formula. Since reasoning at the level of modular arithmetic is often more efficient in practice than reasoning at the level of bits, QE using bit-blasting might not be the best option if the quantifier-eliminated formula is intended to be used in further modular arithmetic level reasoning.

Another technique for eliminating quantifiers from LMCs is converting the LMCs to equivalent constraints in linear arithmetic over integers [13], and then using QE techniques for linear integer arithmetic such as Omega Test [11]. Similarly, automata-theoretic approaches for eliminating quantifiers from linear integer constraints [15] can also be used. However, this approach scales poorly in practice. Moreover, this approach destroys the modular arithmetic structure of the problem since the resulting formula is a linear integer arithmetic formula and converting this formula back to modular arithmetic is often difficult.

1.2.1 Contributions

– We present a bit-precise and practically efficient algorithm for eliminating quantifiers from conjunctions of LMCs. The algorithm is based on a layered approach, whereby sound but incomplete and cheaper layers are invoked first, and expensive but complete layers are called only when required. The cheapest layers are based on simplifications using LMEs and dropping unconstraining LMDs and LMIs from the problem instance. Subsequently we use a Fourier-Motzkin style layer to eliminate quantifiers from conjunctions of LMIs. The final, most expensive and complete layer is based on model enumeration. Experiments indicate that we do not need to invoke the model enumeration based layer on a wide range of benchmarks arising in practice. The experiments also demonstrate effectiveness of our algorithm over alter-

native QE techniques based on bit-blasting and conversion to integer linear arithmetic.

– We present approaches to extend the aforementioned algorithm to eliminate quantifiers from Boolean combinations of LMCs. We introduce a new decision diagram called Linear Modular Decision Diagram (LMDD) that represents Boolean combinations of LMCs, and present algorithms for QE from LMDDs. We then present an SMT solving based approach for QE from Boolean combinations of LMCs, and a hybrid approach that tries to combine the strengths of the LMDD and SMT solving based approaches. Experiments demonstrate the effectiveness of these approaches. The experiments also demonstrate the utility of these approaches in bounded model checking of word-level RTL designs.

1.3 QE for Propositional Logic

In propositional logic, variables can have values true or false. Formulas are constructed using variables, constants (true, false), and Boolean connectives ($\neg$, $\vee$, $\wedge$). Variables and formulas in propositional logic are also called propositional variables and propositional formulas respectively. Using the notation introduced in Section 1.1, propositional logic can be considered as a first-order theory, where domain of variables is $\{\mathsf{true}, \mathsf{false}\}$ and constraints are variables or constants.

We focus on Skolem function-based techniques for QE from formulas in propositional logic. Formally, let $X = \{x_1, \ldots x_n\}$ and $Y = \{y_1, \ldots y_m\}$ be sets of propositional variables. Let $F(X,Y)$ be a propositional formula over the set $X \cup Y$. A Skolem function ψ_i for x_i in $F(X,Y)$ is a formula over the set $X \setminus \{x_i\} \cup Y$ such

that $\exists x_i.F \equiv F[x_i \mapsto \psi_i]$, where $F[x_i \mapsto \psi_i]$ denotes the formula obtained by substituting occurrences of x_i in F with ψ_i. For example, consider the propositional formula $x_1 \wedge y_1$. We claim that the y_1 and true are two Skolem functions for the variable x_1 in $x_1 \wedge y_1$. To verify the claim, observe that (i) $\exists x_1.(x_1 \wedge y_1)$ is equivalent to y_1, (ii) $(x_1 \wedge y_1)[x_1 \mapsto y_1]$ is equivalent to y_1, and (iii) $(x_1 \wedge y_1)[x_1 \mapsto \mathsf{true}]$ is equivalent y_1.

Other than QE, Skolem functions have many important applications. Skolem functions are used as certificates [8] for satisfiable Quantified Boolean Formulas (QBFs) by QBF solvers. The problem of synthesizing a circuit or program [9] that satisfies the specification $Spec(I,O)$, where I is the set of inputs and O is the set of outputs reduces to computing Skolem functions $\psi(I)$ for variables in O in the formula $Spec(I,O)$.

Our primary motivation for studying Skolem function generation comes from the problem of computing disjunctive decompositions of sequential circuits represented as symbolic transition functions in propositional logic [6]. Here, given a sequential circuit, we wish to obtain "component" sequential circuits, each of which has the same state space as the original circuit, but only a single transition going out of every state. Thus, the set of state transitions of the original circuit is the union of the sets of state transitions of the components. Disjunctive decompositions find their applications in efficient reachability analysis [7].

Given a propositional formula $F(X,Y)$, there are techniques [8, 16, 17] for generating Skolem functions for variables in the set X when $\exists X.F(X,Y)$ is valid. These techniques extract Skolem functions from the proof of validity of $\exists X.F(X,Y)$. However in applications such as QE and computation of disjunctive decomposition, we need to generate Skolem functions $\psi(Y)$ for variables in set X irrespective

of the validity of $\exists X.F(X,Y)$. The work in [9] proposes techniques to generate Skolem functions matching predefined candidate function templates. However such techniques are effective only when the set of candidate function templates is known and small.

The work by Jiang in [18] and the work by Trivedi et. al. in [6] propose composition based techniques to compute Skolem functions. Let $F[x_i \mapsto 0]$ be F with occurrences of x_i replaced by false and $F[x_i \mapsto 1]$ be F with occurrences of x_i replaced by true. The work in [18] computes a Skolem function for x_i in F as a Craig interpolant [19] of $F[x_i \mapsto 1] \wedge \neg F[x_i \mapsto 0]$ and $\neg F[x_i \mapsto 1] \wedge F[x_i \mapsto 0]$. The work in [6] observes that $(F[x_i \mapsto 1] \wedge \neg F[x_i \mapsto 0]) \vee (F[x_i \mapsto 1] \wedge F[x_i \mapsto 0] \wedge h) \vee (\neg F[x_i \mapsto 1] \wedge \neg F[x_i \mapsto 0] \wedge g)$ is a Skolem function for x_i in F, where h and g are any propositional formulas with $X \setminus \{x_i\} \cup Y$ as support. However these techniques necessarily require nested compositions, which cause formula blow-up and vulnerability to memory-outs even for medium-sized benchmarks.

In many practical applications, the formula $F(X,Y)$ is given as a conjunction of smaller formulas rather than as a single monolithic formula. Existing algorithms for generation of Skolem functions do not make use of such factored form and consider $F(X,Y)$ as a monolithic formula. We focus on the generating Skolem functions for propositional formulas given in factored form and argue that exploiting the factored form yields significant performance improvements.

1.3.1 Contributions

– We present a SAT solving based CounterExample Guided Abstraction Refinement (CEGAR) style algorithm for generating Skolem functions for propositional formulas that exploits factored representation of the formu-

las. In contrast to the algorithms in literature, our algorithm neither requires proof of satisfiability nor uses memory-intensive compositions. Experiments demonstrate that our algorithm can generate significantly smaller Skolem functions and performs better on large benchmarks than the existing state-of-the-art algorithms. We show that our algorithm is applicable in more generic cases where the input formula $F(X,Y)$ involves uninterpreted predicates, and demonstrate other potential applications of our algorithm.

1.4 Organization of Thesis

The remainder of this thesis is organized as follows. In Chapter 2 we discuss the related works in detail. This includes a survey of existing QE techniques for linear modular arithmetic and propositional logic. We also present QE techniques for the theory of linear real arithmetic and the theory of linear integer arithmetic – two theories that are closely related to the theory of linear modular arithmetic. In Chapter 3 we present our algorithm for eliminating quantifiers from conjunctions of LMCs. We present approaches to extend this algorithm for eliminating quantifiers from Boolean combinations of LMCs in Chapter 4. Our work on generation of Skolem functions for propositional formulas is presented in Chapter 5. We conclude and suggest directions for future work in Chapter 6.

For clarity of exposition, in most of the lemmas/propositions/theorems presented in the following chapters, we give illustrative examples before presenting the detailed proofs. Chapter 3 and Chapter 4 are extended versions of our works in [20, 21], and also reported in [22]. Chapter 5 is an extended version of our work in [23].

Chapter 2

Related Works

In this chapter, we present existing QE techniques for linear modular arithmetic and propositional logic. We also present QE techniques for the theory of linear real arithmetic and the theory of linear integer arithmetic. These theories are closely related to the theory of linear modular arithmetic. Moreover some of the techniques that we present in the following chapters are extensions of QE techniques for these theories.

We will use symbols $\mathcal{T}_{\mathcal{R}}$, $\mathcal{T}_{\mathcal{Z}}$ and $\mathcal{T}_{\mathcal{M}}$ to refer to the theories linear real arithmetic, linear integer arithmetic and linear modular arithmetic respectively. Moreover, we will use "linear arithmetic" when the distinction between the theories $\mathcal{T}_{\mathcal{R}}$, $\mathcal{T}_{\mathcal{Z}}$ and $\mathcal{T}_{\mathcal{M}}$ is not important. QE algorithms for linear arithmetic usually work on conjunctions of constraints. In order to apply these algorithms on formulas that are arbitrary Boolean combinations of constraints, the formula is often converted into disjunctive normal form (DNF) $d_1 \vee \cdots \vee d_m$, where each d_i is a conjunction of constraints. For each d_i, $\exists x.\, d_i$ is computed by using the QE algorithm that works on conjunctions of constraints. Hence, in this chapter, we initially focus on QE

algorithms that work on conjunctions of constraints, and then explore techniques to extend these algorithms to arbitrary Boolean combinations of constraints.

Satisfiability problem (decision problem) for a theory is the problem of deciding if a given formula in the theory is satisfiable or unsatisfiable. Algorithms for satisfiability problem for a theory are called satisfiability procedures (decision procedures). A formula F over variables $x_1, \ldots, x_n$ is satisfiable iff $\exists x_1 \cdots \exists x_n . F$ is true. Hence QE algorithm for a theory can be used as a decision procedure for the theory. In fact the relationship between QE algorithms and decision procedures is even deeper. Advances in decision procedures have always motivated development of efficient QE algorithms. For example SAT based QE algorithms [24, 25, 26] for propositional logic were motivated by the success of SAT solvers. Similarly, many key operations performed by decision procedures are closely related to QE. For example, as observed in [24], resolution, the basic operation used in SAT solving, is actually performing QE. Let c_1, c_2 be two clauses and let c_3 be the resolvent of c_1 and c_2 on a variable x. Then, c_3 is equivalent to $\exists x . (c_1 \wedge c_2)$. Due to this strong connection between QE algorithms and decision procedures, along with QE algorithms, we also provide a brief description of decision procedures for the theories.

The remaining part of this chapter is organized as follows. We explain QE algorithms for conjunctions of constraints in $\mathcal{T}_{\mathcal{R}}$ in Section 2.1 and QE algorithms for conjunctions of constraints in $\mathcal{T}_{\mathcal{Z}}$ in Section 2.2. Existing algorithms for QE from conjunctions of constraints in $\mathcal{T}_{\mathcal{M}}$ are explained in Section 2.3. Section 2.4 describes techniques to extend these algorithms to arbitrary Boolean combinations of constraints. Techniques for QE from propositional logic formulas are explained in Section 2.5. We conclude in Section 2.6.

2.1 Linear Real Arithmetic

$\mathcal{T}_{\mathcal{R}}$ permits linear terms of the form $a_1x_1 + \cdots + a_nx_n + a_0$, where $a_0, \ldots, a_n$ are rational constants and $x_1, \ldots, x_n$ are variables ranging over reals. Constraints in $\mathcal{T}_{\mathcal{R}}$ are of the form $s \bowtie t$ where s, t are linear terms permitted by the theory and $\bowtie \in \{=, \neq, <, \leq\}$ (see [30] for more details on $\mathcal{T}_{\mathcal{R}}$).

We wish to compute $\exists x.C$, where C is a quantifier-free conjunction of constraints in $\mathcal{T}_{\mathcal{R}}$ involving x. We assume that C involves only equalities and strict inequalities. Note that this does not cause any loss of generality, since disequalities and weak inequalities can be replaced by equalities and strict inequalities using the equivalences 1) $(s \leq t) \equiv (s < t) \vee (s = t)$, and 2) $(s \neq t) \equiv (s < t) \vee (s > t)$.

We further assume that the constraints in C are expressed in the *normalized* form $x \bowtie t$, where t is a term free of x and $\bowtie \in \{=, <, >\}$. This is always possible by using the standard arithmetic transformations permitted in real arithmetic. For example, consider the $\mathcal{T}_{\mathcal{R}}$ constraint $(3x + 10 > y)$. It can be equivalently expressed in form $(3x > y - 10)$ by adding -10 to both sides of the inequality. It can be then expressed in the normalized form $(x > \frac{1}{3}y - \frac{10}{3})$ by dividing both sides by 3.

Note that if C contains an equality $x = t$, then $\exists x.C$ can be computed very easily. Substituting the occurrences of x in all other constraints in C by t, and then dropping $x = t$ from C gives $\exists x.C$. Hence, in the remaining part of this section, we will assume that C involves only strict inequalities of the form $x < s$ and $x > t$.

We present three algorithms for QE from such conjunctions of constraints in $\mathcal{T}_{\mathcal{R}}$ - Fourier-Motzkin Algorithm, Ferrante and Rackoff's Algorithm and Loos and Wiespfenning's Algorithm (Although in the following description we focus on only conjunctions of constraints, Ferrante and Rackoff's Algorithm and Loos

and Wiespfenning's Algorithm can directly work on Boolean combinations of constraints in $\mathcal{T}_{\mathcal{R}}$). We then present automata-theoretic approaches for QE from conjunctions of constraints in $\mathcal{T}_{\mathcal{R}}$, and conclude the section with a discussion on decision procedures for $\mathcal{T}_{\mathcal{R}}$ and some complexity results.

2.1.1 Fourier-Motzkin Algorithm

The fundamental idea behind Fourier-Motzkin Algorithm is to eliminate the variable by projecting the constraints on the remaining variables in the conjunction. Let L be the set of constraints in C of the form $(x > t)$, and U be the set of constraints in C of the form $(x < s)$. The result of projecting C on the remaining variables can be computed as $\bigwedge_{(x<s)\in U,\,(x>t)\in L} (t < s)$. As an example, consider the problem of computing $\exists x.C$, where C is the formula $(x < 3y+2) \wedge (x > \frac{1}{2}z + \frac{3}{2})$. Here L is $\{x > \frac{1}{2}z + \frac{3}{2}\}$ and U is $\{x < 3y+2\}$. Projecting $(x < 3y+2) \wedge (x > \frac{1}{2}z + \frac{3}{2})$ on y and z and thereby eliminating x gives $\exists x.C$ as $(\frac{1}{2}z + \frac{3}{2} < 3y+2)$.

2.1.2 Ferrante and Rackoff's Algorithm

Ferrante and Rackoff's Algorithm [31] is a *test point* based QE algorithm. Test point based algorithms express $\exists x.C$ as an equivalent finite disjunction of the form $C[x \mapsto t_1] \vee \cdots \vee C[x \mapsto t_m]$, where $t_1, \ldots, t_m$ are terms on free variables in C, and $C[x \mapsto t_i]$ represents C with occurrences of x replaced by t_i. The terms $t_1, \ldots, t_m$ are called test points, and are generated from the constraints in C.

Let A be the set of constraints in C. Ferrante and Rackoff's Algorithm computes $\exists x.C$ as $C[x \mapsto -\infty] \vee C[x \mapsto +\infty] \vee \bigvee_{(x \bowtie s),(x \bowtie t)\in A} C[x \mapsto \frac{s+t}{2}]$, where $\bowtie \in \{<, >\}$. The occurrences of $-\infty$ and $+\infty$ can be eliminated by using the equiv-

alences $(s < -\infty) \equiv (+\infty < t) \equiv \mathsf{false}$ and $(-\infty < t) \equiv (s < +\infty) \equiv \mathsf{true}$. As an example, consider the problem of computing $\exists x.C$, where C is the formula $(x < 5) \wedge (x > \frac{5}{3})$. Here $C[x \mapsto -\infty]$ and $C[x \mapsto +\infty]$ simplify to false. Since the constraints in C are $(x < 5)$ and $(x > \frac{5}{3})$, the terms s and t take values from the set $\{5, \frac{5}{3}\}$, and $\frac{s+t}{2}$ takes values from the set $\{5, \frac{5}{3}, \frac{5+\frac{5}{3}}{2}\}$. Note that $C[x \mapsto 5]$ and $C[x \mapsto \frac{5}{3}]$ both simplify to false, whereas $C[x \mapsto \frac{5+\frac{5}{3}}{2}]$ simplifies to true. Hence $\exists x.C$ is computed as true.

2.1.3 Loos and Wiespfenning's Algorithm

Loos and Wiespfenning's Algorithm [32] is an optimization of Ferrante and Rackoff's Algorithm. Suppose there are k constraints in C. It can be observed that the number of test points Ferrante and Rackoff's Algorithm examines is $O(k^2)$. Loos and Wiespfenning's algorithm reduces the number of test points to be examined to $O(k)$.

Let L be the set of constraints in C of the form $x > t$. Loos and Wiespfenning's Algorithm computes $\exists x.C$ as $C[x \mapsto -\infty] \vee \bigvee_{(x>t)\in L} C[x \mapsto t + \varepsilon]$, where ε is a positive number close to zero. As in Ferrante and Rackoff's algorithm, $-\infty$ and ε can be eliminated by using the equivalences $(s < -\infty) \equiv \mathsf{false}$, $(-\infty < t) \equiv \mathsf{true}$, $(s + \varepsilon < t) \equiv (s < t)$, and $(s < t + \varepsilon) \equiv (s \leq t)$. As an example, consider again the problem of computing $(x < 5) \wedge (x > \frac{5}{3})$. We have seen that $C[x \mapsto -\infty]$ simplifies to false. The set L is $\{x > \frac{5}{3}\}$. The term $t + \varepsilon$ takes values from the set $\{\frac{5}{3} + \varepsilon\}$, and $C[x \mapsto t + \varepsilon]$ simplifies to true. Hence $\exists x.C$ is computed as true.

2.1.4 Automata-theoretic Approaches

The key idea behind the automata-theoretic approaches is to convert the conjunction C to an automaton such that the language accepted by the automaton corresponds to the solutions of C. The work in [33] shows that constraints in $\mathcal{T}_{\mathcal{R}}$ can be represented using weak deterministic Büchi automata, a restricted class of Büchi automata. QE then reduces to applying projection operation on the automaton for the conjunction. The tools LIRA [34] and LASH [35] provide such automata-theoretic QE implementations. However, there are two fundamental problems with these approaches. First of all, the automaton constructed often blows up in practice. Secondly, the result of QE is an automaton, not a formula, and synthesizing a formula from the automaton is difficult.

2.1.5 Complexity Results and Decision Procedures

The QE algorithms for conjunctions of constraints in $\mathcal{T}_{\mathcal{R}}$ that we saw have doubly exponential worst case complexities. Let m be the number of constraints in C. After the execution of Fourier-Motzkin Algorithm, the number of constraints in $\exists x.C$ can grow to $\frac{m^2}{4}$ in the worst case. Let n be the number of variables to be eliminated. Elimination of n variables using Fourier-Motzkin Algorithm can thus result in $\frac{m^{2^n}}{4^n}$ constraints. As a result, the number of steps involved in the elimination of n variables using Fourier-Motzkin Algorithm is doubly exponential in n in the worst case. Similarly, in the cases of both Ferrante and Rackoff's Algorithm and Loos and Wiespfenning's Algorithm, the worst case time complexity is $2^{2^{l.k}}$, where l is the length of the original conjunction C and k is a fixed positive constant.

The decision problem for conjunctions of constraints in $\mathcal{T}_{\mathcal{R}}$ is solvable in polynomial time as originally shown by Khachian in [36] and later by Karmarkar in [37]. Because of their comparatively high theoretical and practical complexities, the QE algorithms for $\mathcal{T}_{\mathcal{R}}$ are not usually used as decision procedures. The most popular decision procedure used in practice for conjunctions of constraints in $\mathcal{T}_{\mathcal{R}}$ is a variant [38] of Simplex algorithm [39] for linear programming. Although Simplex is exponential in worst case, in practice the need for exponential number of steps occurs very rarely.

2.2 Linear Integer Arithmetic

$\mathcal{T}_{\mathcal{Z}}$ permits linear terms of the form $a_1x_1 + \cdots + a_nx_n + a_0$, where $a_0, \ldots, a_n$ are integer constants and $x_1, \ldots, x_n$ are variables ranging over integers. Constraints in $\mathcal{T}_{\mathcal{Z}}$ are of the form $s \bowtie t$ where s, t are linear terms permitted by the theory and $\bowtie \in \{=, \neq, <, \leq\}$ (see [30] for more details).

$\mathcal{T}_{\mathcal{Z}}$ as defined above does not admit QE. To see this, consider the formula $\exists x.\,(y = 2x)$ in $\mathcal{T}_{\mathcal{Z}}$. This formula expresses the set y of integers that when divided by 2 gives another integer. In other words, this formula expresses the set of even integers. However, it can be observed that there is no formula in $\mathcal{T}_{\mathcal{Z}}$ that can express the set of even integers. To overcome this problem, $\mathcal{T}_{\mathcal{Z}}$ is augmented with *congruence constraints* of the form $(t = 0 \pmod{k})$, where k is a positive integer and t is a linear term. Note that *congruence constraints* are similar to LMEs, but the modulus need not be a power of 2. It can be observed that $(t = 0 \pmod{k})$ is semantically equivalent to $\exists x.\,(t = kx)$. The augmented theory admits QE. For example, the formula $\exists x.\,(y = 2x)$ is equivalent to $(y = 0 \pmod{2})$. As

we are interested in the problem of QE, henceforth we will use $\mathcal{T}_{\mathcal{Z}}$ to denote the augmented theory of linear integer arithmetic.

In this section, we will focus on techniques to compute $\exists x.C$, where C is a quantifier-free conjunction of constraints in $\mathcal{T}_{\mathcal{Z}}$ involving x. We assume that C involves only equalities and strict inequalities. As in the case of reals, this does not cause any loss of generality, since disequalities and weak inequalities can be replaced by strict inequalities using the equivalences 1) $(s \leq t) \equiv (s < t+1)$, and 2) $(s \neq t) \equiv (s < t) \vee (s > t)$.

QE and reasoning techniques for constraints in $\mathcal{T}_{\mathcal{Z}}$ are more involved compared to those for constraints in $\mathcal{T}_{\mathcal{R}}$. The additional difficulty arises primarily due to the reason that unlike reals which are dense, integers are discrete. Recall that equalities and strict inequalities in $\mathcal{T}_{\mathcal{R}}$ can be transformed to the *normalized* form $x \bowtie t$, where t is a term free of x and $\bowtie \in \{=, <, >\}$. However such transformation may not preserve equivalence in case of $\mathcal{T}_{\mathcal{Z}}$ as it can generate non-integral coefficients. Hence a *weaker normalized* form $ax \bowtie t$, where t is a term free of x, $\bowtie \in \{=, <, >\}$, and a is a positive integer is defined for $\mathcal{T}_{\mathcal{Z}}$. For example, consider the constraint $-2x + y > 10$. It can be expressed in the normalized form $2x < y - 10$ by adding $-y$ to both sides and then multiplying both sides by -1.

As in the case of $\mathcal{T}_{\mathcal{R}}$, the presence of equalities in C simplifies the computation of $\exists x.C$ considerably. Let C involves an equality $ax = t$. Let σ be the least common multiple (lcm) of the coefficients of x in the constraints in C. The constraints in C can be multiplied by appropriate constants so that σ is the coefficient of x in all constraints. Let the equality $ax = t$ gets multiplied by b and gets converted to $\sigma x = bt$ in this process, where $b = \frac{\sigma}{a}$. Substituting the occurrences of σx in all constraints in C other than $\sigma x = bt$ by bt, and then replacing

$\sigma x = bt$ by $(bt = 0 \pmod{\sigma})$ gives $\exists x.C$. For example, consider the problem of computing $\exists x.((2x = 3y+2) \wedge (3x > 4z+3))$. Here σ is lcm(2,3) $= 6$. Multiplying $(2x = 3y+2)$ by 3 and $(3x > 4z+3)$ by 2, we have $\exists x.((6x = 9y+6) \wedge (6x > 8z+6))$. Replacing the $6x$ in $(6x > 8z+6)$ by $9y+6$ and simplifying gives $\exists x.(6x = 9y+6) \wedge (9y > 8z)$. This is equivalent to $(9y+6 = 0 \pmod{6}) \wedge (9y > 8z)$. Henceforth, we will assume that C involves only strict inequalities of the form $ax < t$ and $bx > s$.

The pioneering work on QE from conjunctions of constraints in $\mathcal{T}_{\mathcal{Z}}$ was done by Presburger [40]. Here we present a more efficient version of Presburger's QE procedure introduced by Cooper in [41] called Cooper's Algorithm (Similar to Ferrante and Rackoff's Algorithm and Loos and Wiespfenning's Algorithm for $\mathcal{T}_{\mathcal{R}}$, Cooper's Algorithm can directly work on Boolean combinations of constraints. However in the following description we focus only on conjunctions of constraints.). We also present Omega Test Algorithm [11], an extension of Fourier-Motzkin Algorithm for $\mathcal{T}_{\mathcal{Z}}$. We then present automata-theoretic approaches, and conclude the section with a discussion on decision procedures for $\mathcal{T}_{\mathcal{Z}}$ and some complexity results.

2.2.1 Cooper's Algorithm

Cooper's algorithm is a test point based QE algorithm. It expresses $\exists x.C$ as an equivalent disjunction of the form $C[x \mapsto t_1] \vee \cdots \vee C[x \mapsto t_m]$, where $t_1, \ldots, t_m$ are terms on free variables in C.

Let σ be the lcm of the coefficients of x in the constraints in C. The constraints in C are multiplied by appropriate constants so that σ is the coefficient of x in all constraints. Let C' be the formula obtained by replacing the occurrences

of σx in C by a fresh variable x'. Let D be the formula $C' \wedge (x' = 0 \pmod{\sigma})$. It can be observed that $\exists x.C$ is equivalent to $\exists x'.D$. Let L be the set of constraints in D of the form $x' > t$. Now $\exists x.C$ is equivalent to $\bigvee_{j=1}^{\sigma} D[x' \mapsto -\infty + j] \vee \bigvee_{j=1}^{\sigma} \bigvee_{(x'>t) \in L} D[x' \mapsto t + j]$. The occurrences of $-\infty$ are eliminated by using the equivalences $(-\infty + j > t) \equiv \mathsf{false}$ and $(-\infty + j < t) \equiv \mathsf{true}$.

As an example, consider the problem of computing $\exists x.C$, where C is the formula $(3x < 7) \wedge (2x > 3)$. Note that σ is lcm(3, 2) = 6. Multiplying $(3x < 7)$ by 2 and $(2x > 3)$ by 3, we get $\exists x.((6x < 14) \wedge (6x > 9))$. Replacing $6x$ by x', and adding the constraint $(x' = 0 \pmod 6)$, we have $\exists x'.D$, where D is $(x' < 14) \wedge (x' > 9) \wedge (x' = 0 \pmod 6)$. Note that $\bigvee_{j=1}^{\sigma} D[x' \mapsto -\infty + j]$ simplifies to false. Since the set L is $\{x' > 9\}$, t takes the value 9, and $t + j$ takes values from the set $\{10, 11, 12, 13, 14, 15\}$. Note that $\bigvee_{j=1}^{\sigma} \bigvee_{(x'>t) \in L} D[x' \mapsto t + j]$ simplifies to true, since $D[x' \mapsto 12]$ is true. Therefore $\exists x.C$ computed is true.

2.2.2 Omega Test Algorithm

Omega Test Algorithm is an extension of Fourier-Motzkin Algorithm for QE from constraints in $\mathcal{T}_{\mathcal{Z}}$. Although originally presented as a decision procedure for conjunctions of constraints in $\mathcal{T}_{\mathcal{Z}}$, it can be extended to a QE algorithm in a straightforward manner.

For clarity in exposition, let us initially consider the simpler problem of computing $\exists x.(ax < t) \wedge (bx > s)$. First $(ax < t)$ is multiplied by b and $(bx > s)$ is multiplied by a so that ab is the coefficient of x in both the constraints. Thus we have $\exists x.(abx < bt) \wedge (abx > as)$. Applying Fourier-Motzkin Algorithm on $\exists x.(abx < bt) \wedge (abx > as)$ gives $(bt > as)$. However $(bt > as)$ is an over-approximation of

$\exists x.\,(abx < bt) \wedge (abx > as)$, called *real shadow*. Note that if $(bt - as > ab)$ holds, then existence of an integer i such that $(abi < bt) \wedge (abi > as)$ is guaranteed. Thus $(bt - as > ab)$ is an under-approximation of $\exists x.\,(abx < bt) \wedge (abx > as)$. The constraint $(bt - as > ab)$ is called *dark shadow*.

Consider $\exists x.\,(abx < bt) \wedge (abx > as)$ when $(bt - as \leq ab)$. This implies $(as < abx < as + ab)$, i.e., $(s < bx < s + b)$. Hence $\exists x.\,(abx < bt) \wedge (abx > as)$ is equivalent to $\exists x.\ \bigvee_{j=1}^{b-1} (bx = s + j) \wedge (abx < bt) \wedge\ (abx > as)$ when $(bt - as \leq ab)$. The formula $\bigvee_{j=1}^{b-1} (bx = s + j)$ is called *grey shadow*. Substituting the occurrences of bx in $(abx < bt)$ and $(abx > as)$ by $s + j$, and then replacing $(bx = s + j)$ by $(s + j = 0 \pmod b)$ gives $\bigvee_{j=1}^{b-1} (s + j = 0 \pmod b) \wedge (a(s + j) < bt) \wedge\ (a(s + j) > as)$, which can be simplified to $\bigvee_{j=1}^{b-1} (s + j = 0 \pmod b) \wedge (a(s + j) < bt) \wedge\ (bt > as)$.

Putting everything together, $\exists x.\,(ax < t) \wedge (bx > s)$ is equivalent to the disjunction of $(bt - as > ab)$ and $\bigvee_{j=1}^{b-1} (s + j = 0 \pmod b) \wedge (a(s + j) < bt) \wedge\ (bt > as)$. In general, let L be the set of constraints in C of the form $(bx > s)$ and U be the set of constraints in C of the form $(ax < t)$. Then $\exists x.C$ is equivalent to

$$\bigwedge_{(ax<t)\in U,\,(bx>s)\in L} \exists x.\,(ax < t) \wedge (bx > s).$$

As an example consider the problem of computing $\exists x.C$, where C is the formula $(5x < y) \wedge (6x > y)$. Here $a = 5$, $b = 6$ and $s = t = y$. *Real shadow* $(bt > as)$ is $(y > 0)$, and *dark shadow* $(bt - as > ab)$ is $(y > 30)$. The formula $\bigvee_{j=1}^{b-1} (s + j = 0 \pmod b) \wedge (a(s + j) < bt)$ simplifies to $(y = 5 \pmod 6 \wedge y > 5) \vee (y = 4 \pmod 6 \wedge y > 10) \vee (y = 3 \pmod 6 \wedge y > 15) \vee (y = 2 \pmod 6 \wedge y > 20) \vee (y = 1 \pmod 6 \wedge y > 25)$. Hence $\exists x.C$ is computed as $(y > 30) \vee (y = 5 \pmod 6 \wedge y > 5) \vee (y = 4 \pmod 6 \wedge y > 10) \vee (y = 3 \pmod 6 \wedge y > 15) \vee (y = 2 \pmod 6 \wedge y > 20) \vee (y = 1 \pmod 6 \wedge y > 25)$.

2.2.3 Automata-theoretic Approaches

Büchi showed [42] that a conjunction of constraints in $\mathcal{T}_{\mathcal{Z}}$ can be represented by a finite state automaton such that the language accepted by the automaton corresponds to the solutions of the conjunction. A more efficient technique to represent constraints in $\mathcal{T}_{\mathcal{Z}}$ by automaton was proposed by Boudet and Comon in [43], which was further improved by Wolper and Boigelot [44]. Once the conjunction is represented as an automaton, QE can be done by applying projection operation on the automaton. An implementation of the technique by Wolper and Boigelot can be found in the tool LASH [35]. LIRA [34] provides a more efficient implementation of this technique that reduces the number of states in the automaton and thereby keeps the automaton smaller. However, as in the case of $\mathcal{T}_{\mathcal{R}}$, the primary bottleneck of the automaton-based approaches is that the automaton constructed often blows up in practice as observed in [15]. Moreover, deriving a formula from the automaton obtained after QE is difficult.

2.2.4 Complexity Results and Decision Procedures

Let l be the length of the input conjunction of constraints C. The worst case time complexity of Cooper's Algorithm is $2^{2^{2^{l.k}}}$, where k is a fixed positive constant. The time taken by Omega Test Algorithm and the size of quantifier eliminated result can be proportional to the absolute values of the coefficients in the worst case.

The decision problem for conjunctions of constraints in $\mathcal{T}_{\mathcal{Z}}$ is NP-complete [45]. Although Omega Test Algorithm is used as a decision procedure for conjunctions of constraints in $\mathcal{T}_{\mathcal{Z}}$, there exist more popular decision procedures based on

branch-and-bound and Gomory's cutting planes [46]. In both these approaches, given a conjunction C of constraints in $\mathcal{T}_{\mathcal{Z}}$, it is first checked to see if C has a solution when the variables are not required to be integers. This version of C without the integrality requirement is called *relaxed problem*, and can be solved using Simplex algorithm. If the relaxed problem is unsatisfiable, then C is also unsatisfiable. Otherwise let π be the solution returned by Simplex for the relaxed problem. If all variables are assigned integer values in π, then π is a solution to C as well. Otherwise let π assigns a fractional value f_i to variable x_i. In branch-and-bound approach, two conjunctions C_1 and C_2 are created, where C_1 is $C \wedge (xi \leq \lfloor f_i \rfloor)$ and C_2 is $C \wedge (xi \geq \lceil f_i \rceil)$. The original problem of deciding C is now split into subproblems of deciding C_1 and C_2, since C has a solution only if C_1 or C_2 has a solution. In Gomory's cutting planes approach, a new inequality l (called cutting plane) is added to C such that $C \Rightarrow l$ and l avoids the fractional solution π. The problem thus changes to deciding $C \wedge l$.

2.3 Linear Modular Arithmetic

Recall that unlike $\mathcal{T}_{\mathcal{Z}}$, variables in modular arithmetic have finite domain. Moreover, the successor of $2^p - 1$ in modular arithmetic with modulus 2^p is 0. Thus semantics of constraints in $\mathcal{T}_{\mathcal{M}}$ differ from that of constraints in $\mathcal{T}_{\mathcal{Z}}$, and hence QE techniques for $\mathcal{T}_{\mathcal{Z}}$ cannot be directly used for QE from constraints in $\mathcal{T}_{\mathcal{M}}$.

Existence of inverses: Let Z_{2^p} denote the set $\{0, \ldots, 2^p - 1\}$. It can be observed that for each element $c \in Z_{2^p}$, there exits an element $d \in Z_{2^p}$ such that $c + d = 0 \pmod{2^p}$. In other words, additive inverses modulo 2^p exist for all elements in Z_{2^p}. For a term t, we use $-t$ to denote the additive inverse of t modulo 2^p. Multi-

plicative inverses modulo 2^p exist only for odd elements in Z_{2^p}. The work in [74] gives an $O(p)$-time algorithm to compute multiplicative inverses modulo 2^p for odd elements in Z_{2^p}.

Since every variable in an LMC $a_1 \cdot x_1 + \cdots + a_n \cdot x_n \bowtie a_0 \pmod{2^p}$, where $\bowtie \in \{=, \neq, <, \leq\}$, represents a p-bit integer, it follows that a set of LMCs sharing a variable must have the same modulus. Hence we will assume without loss of generality that whenever we consider a conjunction of LMCs sharing a variable, all the LMCs have the same modulus.

The most dominant technique used in practice for eliminating quantifiers from LMCs is conversion to bit-level constraints (also called bit-blasting [13]), followed by bit-level QE. As an example, consider the LMI $(s < x + y)$ with modulus 8. In bit-blasting, the term $x + y$ on the right-hand-side is expressed as an equivalent bit-level formula corresponding to a 3-bit ripple carry adder, and then the LMI is expressed as an equivalent bit-level formula corresponding to a 3-bit comparator. However this technique irretrievably destroys the word-level structure of the problem, and scales poorly as the width of bit-vectors increases.

LMCs can also be expressed as equivalent formulas in $\mathcal{T}_{\mathcal{Z}}$ [83], and then QE algorithms for $\mathcal{T}_{\mathcal{Z}}$ can be used. For example the LMI $(s < x + y)$ with modulus 8 can be equivalently expressed as a $\mathcal{T}_{\mathcal{Z}}$ formula $ite(x + y \leq 7, s = x + y, s = x + y - 8) \wedge (0 \leq x \leq 7) \wedge (0 \leq y \leq 7)$, where *ite* represents if-then-else, and x, y, s are integer variables. However as observed in [13], this approach scales poorly in practice primarily due to the blow-up in formula size during the conversion to $\mathcal{T}_{\mathcal{Z}}$. Moreover, this approach destroys the modular arithmetic structure of the problem since the resulting formula is in $\mathcal{T}_{\mathcal{Z}}$ and converting this formula back to $\mathcal{T}_{\mathcal{M}}$ is often difficult.

In the remaining part of this section, we first discuss the known complexity results for $\mathcal{T}_{\mathcal{M}}$, and then describe decision procedures for $\mathcal{T}_{\mathcal{M}}$.

2.3.1 Complexity Results

Satisfiability problem for a conjunction of LMEs is polynomial-time [79]. However, satisfiability problem for conjunctions of even very limited fragments of LMDs or LMIs are proved to be NP-hard.

Jain et. al in [80] proves that the satisfiability problem for a conjunction of LMDs with modulus 4 is NP-hard. Any instance of 3-SAT problem can be reduced to an instance of satisfiability problem for a conjunction of LMDs with modulus 4 in polynomial-time.

Gange et. al's work in [81] gives a simple reduction from graph 3-colourability problem to the satisfiability problem for a conjunction of constraints of the form $x_1 \neq x_2 \pmod{m}$, where $m \geq 3$. Consider an instance of graph 3-colourability problem $G =< V, E >$, where $V = \{v_1, \ldots, v_n\}$ is the set of vertices and E is the set of edges. For each edge $(v_i, v_j) \in E$, let us introduce a constraint $x_i \neq x_j \pmod{3}$. It can be seen that the conjunction of these constraints is satisfiable iff G is 3-colourable. Since $x_1 \neq x_2 \pmod{m}$ is equivalent to $x_1 - x_2 \geq 1 \pmod{m}$, this reduction applies also for a conjunction of constraints of the form $x_1 - x_2 \geq 1 \pmod{m}$ with $m \geq 3$.

The work in [12] introduces Modular Difference Logic (MDL) constraints. MDL constraints are a fragment of LMIs of the form $x_1 + k_1 \leq x_2 + k_2 \pmod{2^p}$, where x_1, x_2 are variables, and k_1, k_2 are constants. The work in [12] proves that the satisfiability problem for conjunctions of MDL constraints of the form $x_1 + 1 \leq x_2 \pmod{2^p}$ or of the form $x_1 \leq x_2 + 2^p - 1 \pmod{2^p}$ with $2^p \geq 4$ is NP-hard.

Any instance of graph 3-colourability problem can be reduced to an instance of satisfiability problem for a conjunction of such constraints in polynomial-time.

Given a conjunction C of LMCs over variables $x_1, \ldots, x_n$, the problem of computing $\exists x_1 \cdots \exists x_t . C$ is NP-hard, since (i) the satisfiability problem for a conjunction of LMCs is NP-hard even when the modulus is a priori fixed and (ii) C is satisfiable iff $\exists x_1 \cdots \exists x_n . C$ is true.

A related, but possibly simpler problem is quantifying a single variable from a conjunction of LMCs. Note that the above NP-hardness result for quantifying a set of variables from a conjunction of LMCs does not imply NP-hardness of the problem for a single variable. However, even the problem for a single variable is NP-hard when the modulus is not a priori fixed but part of the input and the modulus is not necessarily a power of 2. Given a collection of ordered pairs $\{(a_1, b_1), \ldots, (a_n, b_n)\}$, where a_i, b_i are positive integers with $a_i \leq b_i$ for $1 \leq i \leq n$, the problem of computing $\exists x. ((x \neq a_1 \pmod{b_1}) \wedge \ldots \wedge (x \neq a_n \pmod{b_n}))$ is NP-hard as shown in [45].

2.3.2 Decision Procedures

There are several works (see [75, 76]) on solving conjunctions of LMEs using variants of Gaussian elimination. The works in [77] and [78] give *Gaussian elimination based* algorithms for deriving solved form for conjunctions of LMEs. The solved form captures all possible solutions of the given conjunction of LMEs. Ganesh et. al in [79] gives a *solve-and-substitute kind* algorithm to derive solved form for conjunctions of LMEs. Given a conjunction of LMEs, the algorithm in [79] initially solves for the variables appearing with odd coefficients. If there is a variable appearing with odd coefficient in an LME, then that LME is chosen.

The term involving the variable appearing with odd coefficient is isolated on the left-hand-side of the LME, and the LME is multiplied by the multiplicative inverse of the variable's coefficient. The variable is then eliminated from other LMEs by substitution. If there are no variables appearing with odd coefficients, then the algorithm selects the variable with coefficient that has the minimum number of factors of 2. Let x be the variable selected this way and let f be the number of factors of 2 in its coefficient. The whole conjunction is divided by 2^f to obtain a conjunction of LMEs with modulus 2^{p-f} which is equisatisfiable with the original conjunction. The variable selected x appears with odd coefficient in the new conjunction of LMEs, and the algorithm proceeds by solving for this variable.

Most of the SMT solvers [82, 10] decide the satisfiability of conjunctions of LMDs and/or LMIs by bit-blasting followed by SAT solving. However because of the bitwidth-dependent blow-up during bit-blasting, this approach suffers from scaling problems for problem instances with large moduli. The SMT solver MathSAT [84] converts LMCs to equivalent formulas in $\mathcal{T}_{\mathcal{Z}}$ and then uses solvers for $\mathcal{T}_{\mathcal{Z}}$. However, as mentioned earlier, this approach suffers from blow-up in formula size during the conversion to $\mathcal{T}_{\mathcal{Z}}$.

Hadarean et. al. in [85] proposes an extension of the well-known congruence closure algorithm [13] for deciding the satisfiability of conjunctions of LMDs. Recall that the standard congruence closure algorithm assumes that variables have infinite domain, and hence it cannot be directly used for solving conjunctions of LMDs. Given a conjunction of LMDs, for example, $(x \neq y) \wedge (x \neq z)$ with modulus 2, the work in [85] initially puts the variables x, y, and z in different congruence classes. It is then checked to see if each congruence class can be assigned a distinct constant value. If this is possible, then we have a satisfying

assignment. In the example, this is not possible since there are three congruence classes and only two distinct constant values with modulus 2. In such cases, the problem is split into a number of sub-problems. In each sub-problem, a different pair of congruence classes are merged. The process is repeated until either the splits lead to inconsistency or a satisfying assignment is obtained. In the example, the algorithm identifies that congruence classes for y and z can be merged to obtain a satisfying assignment for $(x \neq y) \wedge (x \neq z)$.

The work in [85] also proposes an algorithm to decide the satisfiability of conjunctions of a special class of MDL constraints of the form $x_1 \lhd x_2 \pmod{2^p}$, where $\lhd \in \{<, \leq\}$. Note that these constraints do not have wrap-around behaviour. Given a conjunction I of these constraints the algorithm builds the least model M of I by incrementally processing the constraints in I. Least model is the model where all the variables have the least possible values. Note that every such satisfiable conjunction has a least model. Initially $M(x_i)$ is set to 0 for each variable x_i present in I. Let C be the conjunction of constraints that are already processed. Given a new constraint $x \lhd y$, it is attempted to extend M such that M becomes the least model of $C \wedge x \lhd y$. Therefore if the values $M(x)$ and $M(y)$ do not satisfy $x \lhd y$, then $M(y)$ is increased so that $x \lhd y$ is satisfied. This may violate the previously satisfied constraints in C, and may require changing the values of other variables. If M cannot be extended to become the least model of $C \wedge x \lhd y$, then $C \wedge x \lhd y$ is unsatisfiable, and hence I is unsatisfiable.

Gange et. al's work in [81] proposes a sound heuristic to check the satisfiability of MDL constraints. It makes use of a variant of Floyd-Warshall all-pairs shortest path algorithm to derive the relations between all pairs of variables. The relation between variables x_1 and x_2 is over-approximated by a constraint of the

form $x_1 - x_2 \in [k_1, k_2]$, where k_1, k_2 are constants. $[k_1, k_2]$ is called a "wrapped interval" which represents the set $\{d | k_1 \leq d \leq k_2\}$ if $k_1 \leq k_2$ and $\{d | k_1 \leq d \leq 2^p - 1\} \cup \{d | 0 \leq d \leq k_2\}$ if $k_1 > k_2$. If constraints $x_1 - x_2 \in [a, b]$ and $x_1 - x_2 \in [c, d]$ are derived where $[a, b]$ and $[c, d]$ are disjoint wrapped intervals, then the algorithm soundly infers unsatisfiability.

The idea of wrapped intervals (also called clockwise intervals) was actually introduced by Gotlieb et. al. in [86] in the context of program analysis. In this work, the possible values that program variables can take are abstracted by clockwise intervals. Optimal clockwise intervals, i.e., tightest over-approximations of possible values expressed as clockwise intervals, are computed for each program variable. Techniques for computing the optimal clockwise intervals when the program involves (linear modular) arithmetic operations are also proposed.

Modern SMT solvers, such as, Z3 [10] and theorem-provers such as PVS [60] use specialized heuristics [98] to solve quantified bit-vector formulas by Skolemization followed by use of appropriate choices of Skolem functions. The use of p-adic expansions [61, 62] is explored in [63, 64] to solve *non-linear* modular equations. Bruttomesso et al. [110] present a polynomial time algorithm for solving conjunctions of constraints in the core bit-vector theory consisting of only equalities, extractions and concatenations. Their algorithm first generates an equisatisfiable conjunction of equalities on non-overlapping slices of variables involved in the constraints. Congruence closure algorithm is then used for checking the satisfiability of this conjunction of equalities on non-overlapping slices. Similar slicing based ideas for solving conjunctions of bit-vector constraints can be found in [65, 66].

Jain et al. [80] give a polynomial-time algorithm for computing Craig inter-

polants for conjunctions of LMEs. Griggio [67] presents a layered framework for computing interpolants for bit-vector formulas that tries to keep the word-level structure of the problem as much as possible. The cheaper layers use interpolation in EUF (equality + uninterpreted functions) and interpolation by equality substitution. The more expensive layers use conversion to $\mathcal{T}_{\mathcal{Z}}$ and bit-blasting.

2.4 Boolean Combinations

In the QE algorithms we have seen so far, focus was on eliminating quantifiers from conjunctions of constraints. However formulas arising in many practical applications are arbitrary Boolean combinations of constraints, not necessarily conjunctions. In this section, we will explore techniques for extending the QE algorithms for conjunctions of constraints to arbitrary Boolean combinations of constraints.

Ferrante and Rackoff's Algorithm, Loos and Wiespfenning's Algorithm and Cooper's Algorithm which are test point based algorithms, can be directly applied on arbitrary Boolean combinations of constraints. However, the scalability of these algorithms in practice often depends on the underlying representation of the Boolean skeletons of formulas and implementation heuristics used. On the other hand, projection based algorithms such as Fourier-Motzkin and Omega test cannot be directly applied on arbitrary Boolean combinations of constraints. As mentioned earlier, the input formula is first transformed into DNF $d_1 \vee \cdots \vee d_m$, where each d_i is a conjunction of constraints. These conjunctions of constraints are also called *monomes*. For each monome d_i, $\exists x.\, d_i$ is then computed using Fourier-Motzkin / Omega test. Efficient techniques for generation of DNF are

crucial in extending these algorithms to arbitrary Boolean combinations of constraints.

In the following discussion, we will initially focus on techniques for scalable extension of test point based algorithms to arbitrary Boolean combinations of constraints. We will then present efficient techniques for generation of DNF from arbitrary Boolean combinations of constraints.

2.4.1 Extending Test Point Based Algorithms

LinAIG tool [47] implements Loos and Wiespfenning's Algorithm using a data structure called LinAIG. The constraints are abstracted by Boolean variables to obtain Boolean skeletons of formulas. FRAIGs [48] are used to represent the Boolean skeletons, and a map is maintained between the constraints and the Boolean variables. Moreover, they use Craig interpolants [19] to identify and remove redundant constraints generated during application of Loos and Wiespfenning's Algorithm.

Bjørner's work in [49] avoids application of substitutions in the formulation of Loos and Wiespfenning's Algorithm and Cooper's Algorithm. The effect of substitutions is encoded as an additional constraint called *pivot* which is conjoined with the input formula F. Satisfying assignments to $F \wedge pivot$ are generated using a DPLL($\mathcal{T}$) framework, which are then generalized to disjuncts in the formulation of Loos and Wiespfenning's / Cooper's Algorithm. This helps in avoiding $\mathcal{T}$-inconsistent disjuncts in the formulation and unnecessary blow-up in formula size.

Nipkow's work [50] provides implementations of Ferrante and Rackoff's algorithm, Loos and Wiespfenning's algorithm, and Cooper's algorithm that are verified in the theorem prover Isabelle.

Komuravelli et al. [58] introduce model based projection that involves computing model-based under-approximations of existentially quantified formulas. Their work also gives procedures for computing such under-approximations for existentially quantified formulas in linear arithmetic as disjuncts in the formulation of Loos and Wiespfenning's algorithm or Cooper's algorithm. Bjørner et al. [59] give an algorithm that makes use of model based projections for deciding the satisfiability of quantified linear arithmetic formulas. Their algorithm conceptually works as a two-player satisfiability game and can be extended for QE from linear arithmetic formulas.

2.4.2 Generation of DNF

Cavada et al.'s work [27] addresses the problem of existentially quantifying out all numeric variables from formulas involving $\mathcal{T}_{\mathcal{R}}$ constraints and Boolean variables. Their work uses BDDs [51] to represent Boolean structure of the formulas. QE is done by recursively traversing the BDD, carrying along each path, the $\mathcal{T}_{\mathcal{R}}$ constraints encountered on it so far (called the context). Paths with $\mathcal{T}_{\mathcal{R}}$-inconsistent contexts are removed. Because of the dependence of the result of a recursive call on the context, if the same BDD node is encountered following two different paths, the results of the calls are not the same in general. Hence this procedure is not amenable to dynamic programming usually employed in the implementation of BDD operations. In particular, the number of recursive calls in the worst-case is linear in the number of paths, and not the number of nodes, of the original BDD.

The work in [28] presents an efficient algorithm for QE from formulas in the theory of Octagons (a fragment of $\mathcal{T}_{\mathcal{Z}}$ for which Fourier-Motzkin is sufficient for conjunction-level QE). This work introduces decision diagrams for linear arith-

metic called LDDs. The Boolean skeletons of formulas are represented by BDDs and constraints are managed using a separate library external to the BDD package. QE from LDDs makes use of an algorithm called *white-box-QELIM* that eliminates a single variable from an LDD and returns the result as an LDD. Suppose we wish to compute the result of quantifying out a variable x from an LDD rooted at a node f. Let p be the constraint labeling f, and let l and h respectively be the LDDs appearing as the low child and high child of f. *white-box-QELIM* first computes an LDD l' obtained by adding to each 1-path π_l in l the result of quantifying out the variable x from the conjunction of π_l and $\neg p$. Similarly it computes an LDD h' obtained by adding to each 1-path π_h in h the result of quantifying out x from the conjunction of π_h and p. It then recursively calls *white-box-QELIM* on the LDDs l' and h', and returns the disjunction of the LDDs resulting from these calls. Since the result of a recursive call is context-independent, *white-box-QELIM* can be implemented with dynamic programming. This results in considerable performance improvement as reported in [28].

Suppose we wish to quantify out a set of variables X from a $\mathcal{T}_{\mathcal{R}}$ formula F. A straightforward algorithm to do this is an All-SMT algorithm (also called All-SMT loop) that works as follows. An SMT solver call is used to check if F is satisfiable. If F is unsatisfiable, then $\exists X.F$ is false. Otherwise, the solution of F is generalized to a monome C_1 such that $C_1 \Rightarrow F$. The SMT solver is now called to check if $F \wedge \neg C_1$ is satisfiable. If $F \wedge \neg C_1$ is unsatisfiable, then $\exists X.F$ is equivalent to $\exists X.C_1$. Otherwise, the solution of $F \wedge \neg C_1$ is generalized to a monome C_2 such that $C_2 \Rightarrow F$. This loop is repeated until the formula given to the SMT solver becomes unsatisfiable. Each iteration i of the loop generates a monome C_i such that $C_i \Rightarrow F$, for $1 \leq i \leq n$. Finally $\exists X.F$ is equivalent to $\exists X.C_1 \vee \cdots \vee \exists X.C_n$.

The work by Lahiri et al. [54] improves the All-SMT algorithm by considering $\neg C_i$ as a conflicting clause and then performing conflict-driven back-jumping inside the SMT solver. The work by Monniaux in [29] improves the All-SMT algorithm in the following ways. Instead of $\neg C_i$, $\neg \exists X . C_i$ is conjoined with the formula given to the SMT solver. Monniaux calls this "interleaving projection and model enumeration". It is observed that this helps in pruning the solution space of the problem. This results in early termination of the algorithm and reduces the number of SMT solver calls required. Secondly, an SMT solver based procedure is used to further generalize C_i by dropping unnecessary constraints from C_i before $\exists X . C_i$ is computed. This optimization improves the overall performance of the algorithm. Generalizing C_i reduces the time to compute $\exists X . C_i$, and results in generalized $\exists X . C_i$. A generalized $\exists X . C_i$ increases the size of solution space pruned by conjoining $\neg \exists X . C_i$ with the formula given to the SMT solver.

The later work by Monniaux in [52] improves the above algorithm in handling of quantifier alternations. When applied on a $\mathcal{T}_{\mathcal{R}}$ formula with quantifier alternations, for example, $\exists x_1 . \forall x_2 . \exists x_3 . F$, the above algorithm computes the DNF for $\exists x_3 . F$, and then CNF for $\forall x_2 . \exists x_3 . F$, and finally DNF for $\exists x_1 . \forall x_2 . \exists x_3 . F$. The work in [52] proposes a lazy algorithm for computing $\exists x_1 . \forall x_2 . \exists x_3 . F$ that avoids the construction of the full CNFs and DNFs. The algorithm computes under-approximations of $\exists x_1 . \forall x_2 . \exists x_3 . F$ as monomes until the disjunction of these monomes is equivalent to $\exists x_1 . \forall x_2 . \exists x_3 . F$. In order to compute an under-approximation of $\exists x_1 . \forall x_2 . \exists x_3 . F$, initially an under-approximation of $\neg \exists x_3 . F$ is computed. This is then used to compute an under-approximation of $\forall x_2 . \exists x_3 . F$, which is finally used to compute the under-approximation of $\exists x_1 . \forall x_2 . \exists x_3 . F$. Phan et al's work in [53] presents a more general version of this algorithm and extends it to $\mathcal{T}_{\mathcal{Z}}$ formulas.

Techniques for finding generalized implicants are crucial in scalable application of the All-SMT algorithm. Many interesting approaches are proposed recently for deriving such generalized implicants from a given solution of an SMT formula. De Moura et al. [55] present a variation of Boolean constraint propagation in order to identify constraints whose truth values are not essential for determining the satisfiability of a formula. Déharbe et al. [56] present algorithms for generating prime implicants from solutions of formulae by iterative removal of assignments that are not necessary. Niemetz et al. [57] present a dual propagation based technique to extract partial solutions from "full" solutions of SMT formulas. Given a solution m of a formula F, the assignments to variables in m are presented as assumptions to a dual solver which maintains $\neg F$. The assumptions that are inconsistent with $\neg F$ identify the assignments sufficient to satisfy F.

The work by Veanes et al. [87] focuses on automatically constructing monadic decompositions of formulas in quantifier free fragments of first order logic. Monadic decomposition involves transforming a given formula into an equivalent Boolean combination of unary predicates. Veanes et al. give an algorithm for constructing monadic decompositions in Disjunctive Normal Form (DNF). Once such a decomposition is constructed, QE can be achieved by distributing the existential quantifiers over disjunctions in the DNF. This effectively reduces the problem of eliminating quantifiers from a general formula to the problem of eliminating quantifiers from conjunctions involving only unary predicates.

2.5 Propositional Logic

Satisfiability problem for propositional logic has a wide range of practical applications ranging from formal verification [4] to planning in artificial intelligence [88] and from equivalence checking [89] to haplotype inference in bioinformatics [90]. This has led to immense research in this field and to development of SAT solvers that can solve propositional logic formulas involving millions of variables. Most SAT solvers use variants of Davis-Putnam-Loveland-Longemann (DPLL) framework [91]. DPLL framework makes use of a recursive algorithm. Each step in the algorithm involves assigning a value to a variable in the formula, and then checking if this assignment leads to a conflict. If the conflict exists in the formula regardless of the assignments to the variables, then the formula is declared unsatisfiable. Otherwise, if the conflict exists only under the present set of assignments to the variables, then the algorithm tries to learn from the conflict. It then backtracks and undoes some of the assignments it made earlier which led to the conflict. This is repeated until either the formula is declared unsatisfiable or all variables are assigned values in which case the formula is satisfiable.

The interest in propositional logic has also led to well-developed data-structures for representing propositional logic formulas. BDDs [51] provide canonical representation of propositional logic formulas. In applications where canonicity is not required, formulas are often represented as And-Inverter-Graphs (AIGs) [99]. More compact representations such as FRAIGs [48] are also used in specific applications [92].

In the remainder of this section, we will focus on techniques for QE from propositional logic formulas. As mentioned earlier, one of the approaches to perform QE is to express the existentially quantified variables as Skolem functions of

other variables in the formula. QE can be done by substituting the occurrences of the existentially quantified variables in the formula by their Skolem functions. In Subsection 2.5.2 we will present techniques for generating Skolem functions for formulas in propositional logic.

2.5.1 Quantifier Elimination

Suppose we wish to compute $\exists x.F$, where F is a quantifier-free propositional logic formula involving x. It can be observed that $\exists x.F$ is equivalent to $F[x \mapsto 0] \vee F[x \mapsto 1]$, where $F[x \mapsto 0]$ is F with occurrences of x replaced by false and $F[x \mapsto 1]$ is F with occurrences of x replaced by true. Once F is represented as a BDD or an AIG, $F[x \mapsto 0] \vee F[x \mapsto 1]$ can be computed using the standard BDD or AIG operations. The fundamental bottleneck in this technique is that BDDs as well as AIGs blow-up in practice after a number of such QE steps.

The advance in the field of SAT solvers led to the development of SAT-based QE techniques for propositional logic. Since SAT solvers work on Conjunctive Normal Form (CNF), before applying these techniques, the input formula is transformed to equisatisfiable CNF using Tseitin's encoding [100]. Suppose we wish to compute $\exists X.F$, where F is a propositional logic formula in CNF and X is a subset of variables in its support. The work in [24] modifies the DPLL framework to enumerate models of F. Each time a model π of F is obtained, it is generalized to obtain a conjunction of literals (also called cube/implicant) c such that $\pi \Rightarrow c$ and $c \Rightarrow F$. Finally F is equivalent to the disjunction of the implicants generated. Then $\exists X.F$ is obtained by removing the literals x and $\neg x$ from the implicants, where $x \in X$.

The work in [25] also proposes a model enumeration based algorithm to com-

pute $\exists X.F$. However, rather than modifying the DPLL framework, the algorithm in [25] uses queries to an external SAT solver for model enumeration. The SAT solver queries are constrained such that the implicants generated are the shortest. Moreover, the SAT solver queries are incremental, and an incremental SAT solver is used to improve the overall performance of the algorithm.

The recent work by Goldberg et. al. in [26] proposes an alternate SAT solver based QE algorithm. This work makes use of the insight that resolution performed during SAT solving involves QE. Let c_1, c_2 be two clauses and let c_3 be the resolvent of c_1 and c_2 on a variable x. Then, c_3 is equivalent to $\exists x.(c_1 \wedge c_2)$. Given $\exists X.F$, the algorithm proposed in [26] adds resolvent clauses on the variables in X to F. After adding a sufficient number of resolvent clauses, all the clauses containing the variables in X become redundant in $\exists X.F$. These clauses are then dropped and the resulting formula is equivalent to $\exists X.F$.

2.5.2 Skolem Functions

Let $X = \{x_1, \ldots x_n\}$ and $Y = \{y_1, \ldots y_m\}$ be sets of propositional variables. Let $F(X,Y)$ be a propositional formula over the set $X \cup Y$. As mentioned in Chapter 1, a Skolem function ψ_i for x_i in $F(X,Y)$ is a formula over the set $X \setminus \{x_i\} \cup Y$ such that $\exists x_i.F \equiv F[x_i \mapsto \psi_i]$, where $F[x_i \mapsto \psi_i]$ denotes the formula obtained by substituting occurrences of x_i in F with ψ_i.

Given a propositional formula $F(X,Y)$, there are many interesting techniques for generating Skolem functions for variables in the set X when $\exists X.F(X,Y)$ is valid. The pioneering work in this field is by Benedetti in [8]. This work gives a technique for extracting Skolem functions from the proof of validity of $\exists X.F(X,Y)$ generated by the Skolemization-based QBF solver Skizzo [93]. Skizzo

is extended such that the steps to prove the validity are recorded in a *proof log*, which is then read by a tool that constructs Skolem functions encoded as BDDs.

The work in [94] instruments the BDD-based QBF solver EBDDRES so that it generates Skolem functions. To generate Skolem function for an existentially quantified variable x, all the clauses containing x are collected and a BDD f for the conjunction of these clauses is built. Without loss of generality, let x be the variable labeling the root node of f and let l and h respectively be the BDDs appearing as the low child and high child of the root node of f. EBDDRES uses the BDD h as the Skolem function for x. Moreover, it is observed in [94] that negation of l can also be used as the Skolem function for x.

Given a propositional formula $F(X,Y)$, the work by Balabanov and Jiang in [16] proposes a technique to extract Skolem functions for variables in the set X from the cube-resolution proof of validity of $\exists X.F(X,Y)$. When $\exists X.F(X,Y)$ is not valid, this work generates Herbrand functions for variables in the set Y from the clause-resolution proof of invalidity of $\exists X.F(X,Y)$. Herbrand functions for variables in the set Y are simply values of the variables in Y for which $\exists X.F(X,Y)$ is false. Thus Herbrand functions can be used as certificates for the invalidity of $\exists X.F(X,Y)$. This work is applicable to a large class of popular DPLL based QBF solvers such as depQBF [106], QuBE-cert [95], yQuaffle [96] etc., which can generate resolution proofs without much overhead.

Huele et. al.'s work in [97] presents QRAT proof system that captures the pre-processing techniques used by QBF-solvers. The preprocessor bloqqer for QBF solvers is modified so that it generates QRAT proofs. Their recent work in [17] gives techniques to extract Skolem functions from QRAT proofs. Given a propositional formula $F(X,Y)$, this work thus helps in extracting Skolem functions for

variables in the set X if (i) $\exists X.F(X,Y)$ is valid and (ii) the validity can be established only by preprocessing.

Srivastava et. al. in [9] makes use of templates to generate Skolem functions for variables in bit-vector formulas. Given a bit-vector formula $F(X,Y)$, the occurrences of the variables $x_i \in X$ in F are replaced by uninterpreted Skolem functions matching a given template. For example, consider the problem of computing a Skolem function for the variable x in the formula $(x \leq y)$, where x and y are bit-vectors of width, say 3 bits. Using the template $c_1 \cdot y + c_2$ for x, where c_1, c_2 are uninterpreted bit-vector constants of 3 bits, the formula is converted to $\exists c_1.\exists c_2.\forall y.(c_1 \cdot y + c_2 \leq y)$. This formula of the form $\exists C.\forall Y.G(C,Y)$, where C denotes the set of uninterpreted bit-vector constants introduced as above, is solved for the values of variables in C. Instantiating the templates using these values for variables in the set C gives Skolem functions for variables $x_i \in X$ in F matching the template.

The work in [98] makes use of the aforementioned idea to devise a solver for quantified bit-vector formulas with uninterpreted functions. To solve a formula of the form $\forall Y.F(Y)$, where $F(Y)$ is a bit-vector formula with uninterpreted functions over variables in set Y, initially the occurrences of the uninterpreted functions are replaced by templates. Similar to the work in [9], this gives a formula of the form $\exists C.\forall Y.G(C,Y)$, where C denotes the set of uninterpreted bit-vector constants introduced by the templates. The universal quantifiers in $\exists C.\forall Y.G(C,Y)$ are instantiated heuristically in order to convert it to a formula involving only existential quantifiers which is easier to solve. The instantiations of the universal quantifiers are then refined in a counterexample guided manner until either the solutions are obtained or the formula is found to be unsatisfiable modulo the given

templates.

Given a propositional formula $F(X,Y)$ and variable $x_i \in X$, the work in [18] computes Skolem function for x_i in F as a Craig interpolant [19] of $F[x_i \mapsto 1] \wedge \neg F[x_i \mapsto 0]$ and $\neg F[x_i \mapsto 1] \wedge F[x_i \mapsto 0]$, where $F[x_i \mapsto 0]$ denotes F with occurrences of x_i replaced by false and $F[x_i \mapsto 1]$ denotes F with occurrences of x_i replaced by true. The work in [6] observes that $(F[x_i \mapsto 1] \wedge \neg F[x_i \mapsto 0]) \vee (F[x_i \mapsto 1] \wedge F[x_i \mapsto 0] \wedge h) \vee (\neg F[x_i \mapsto 1] \wedge \neg F[x_i \mapsto 0] \wedge g)$ is a Skolem function for x_i in F, where h and g are any propositional formulas with $X \setminus \{x_i\} \cup Y$ as support. When interpolants are used as Skolem functions as suggested in [18], the performance crucially depends on the size of the interpolants and time to compute the interpolants. However it is observed in our experiments that computing interpolants is often time-intensive and interpolants generated by SAT solvers are often not succinct. Moreover these techniques necessarily require nested compositions, which cause formula blow-up and vulnerability to memory-outs even for medium-sized benchmarks.

2.6 Conclusions and Directions of Research

In this chapter, we presented a survey QE techniques for linear arithmetic and propositional logic. This survey led us to the following conclusions and research directions.

1. Existing QE algorithms for conjunctions of LMCs are based on either bit-blasting the constraints or conversion of the constraints to linear integer arithmetic. These techniques not only suffer from performance issues but also destroy the word-level structure of the problem. Development of practi-

cally efficient QE algorithms for conjunctions of LMCs that keep the word-level structure of the constraints is a motivating research direction.

2. It is interesting to see how we can extend such QE algorithms for conjunctions of LMCs to arbitrary Boolean combinations of LMCs in a practically scalable manner. This often requires transformation of the arbitrary Boolean combination of constraints to DNF. Such transformation is usually done using decision diagrams or SMT solving. The primary bottleneck in DNF finding algorithms that make use of SMT solving is usually high computation time. The challenges in DNF finding algorithms that make use of decision diagrams are (i) keeping the diagrams compact and (ii) exploiting reuse of results through dynamic programming. It is interesting to observe if we can combine the strengths of the decision diagram based and SMT solving based algorithms to get best of both worlds.

3. Given a propositional formula $F(X,Y)$, there are techniques for generating Skolem functions for variables in the set X when $\exists X.F(X,Y)$ is valid. Similarly there are template-based techniques for Skolem function generation that can be used when the Skolem function templates are small and are known a-priori. Composition based techniques for Skolem function generation are applicable irrespective of the validity of $\exists X.F(X,Y)$. However these techniques often suffer from memory blow-up in practice. Thus development of efficient algorithms for generating succinct Skolem functions for existentially quantified variables in propositional logic formulas is another motivating research direction.

Chapter 3

Quantifier Elimination for Conjunctions of Linear Modular Constraints

This chapter describes our work on QE from conjunctions of LMCs. Recalling the definition of LMCs in Section 1.2, let p be a positive integer constant, $x_1, \ldots, x_n$ be p-bit non-negative integer variables, and $a_0, \ldots, a_n$ be integer constants in $\{0, \ldots, 2^p - 1\}$. A linear term over $x_1, \ldots, x_n$ is a term of the form $a_1 \cdot x_1 + \cdots a_n \cdot x_n + a_0$, where $\cdot$ denotes multiplication modulo 2^p and $+$ denotes addition modulo 2^p. An LME is a constraint of the form $t_1 = t_2 \pmod{2^p}$, where t_1 and t_2 are linear terms over $x_1, \ldots, x_n$. An LMD is a constraint of the form $t_1 \neq t_2 \pmod{2^p}$, and an LMI is a constraint of the form $t_1 \bowtie t_2 \pmod{2^p}$, where $\bowtie \in \{<, \leq\}$.

The problem we wish to solve in this chapter can be formally stated as follows. Let A denote a conjunction of LMCs over variables $x_1, \ldots, x_n$. We wish to compute

a Boolean combination of LMCs, say φ, such that $\varphi \equiv \exists x_1 \cdots \exists x_t . A$. As observed in Section 2.3.1, this problem is NP-hard. Existing algorithms for solving this problem based on bit-blasting and conversion to linear integer arithmetic suffer from scaling issues and destroy the word-level structure of the problem.

Contributions: We present a bit-precise and practically efficient QE algorithm for conjunctions of LMCs. Our QE algorithm is based on a layered approach. Sound but incomplete and cheaper layers are invoked first, whereas expensive but complete layers are called only when required. The cheapest layer is based on simplification of the problem instance using LMEs. This is followed by a technique that identifies and drops unconstraining LMDs and LMIs from the problem instance, and a Fourier-Motzkin style technique to eliminate quantifiers from conjunctions of LMIs. Finally we use model enumeration as the last resort. Experiments indicate the importance of our layered approach – cheaper layers eliminate a major fraction of quantifiers and model enumeration is not needed on a wide range of benchmarks arising in practice. The experiments also demonstrate effectiveness of our algorithm over existing QE techniques based on bit-blasting and conversion to integer linear arithmetic.

Among the approaches mentioned in Section 2.3 for reasoning about LMCs, the work that is most closely related to our work is that of Ganesh et. al in [79]. The authors of [79] present a technique for reducing LMEs to a solved form by selecting variables in a specific order. While this does not directly give us a technique to eliminate a user-specified variable from a conjunction of LMEs, their work can be extended to achieve this. More importantly, [79] does not consider the problem of eliminating variables from conjunctions involving LMDs or LMIs. This problem is addressed in our work.

3.1 Preliminaries

We will initially focus on the simpler problem of existentially quantifying a single variable from a conjunction of LMCs. For clarity of exposition, we use x to denote the variable to be quantified.

To simplify notation, we assume that all LMCs have modulus 2^p for some positive integer p, unless stated otherwise. We use letters x, y, z, x_1, $x_2, \ldots$ to denote variables, use a, a_1, $a_2, \ldots$, b, b_1, $b_2, \ldots$ to denote constants, and use s, s_1, $s_2, \ldots$, t, t_1, $t_2, \ldots$ to denote linear terms. The letters d, d_1, $d_2, \ldots$ are used to denote LMDs, l, l_1, $l_2, \ldots$ are used to denote LMIs, and c, c_1, $c_2, \ldots$ are used to denote LMCs. Furthermore, we use D, D_1, $D_2, \ldots$ to denote conjunctions of LMDs, I, I_1, $I_2, \ldots$ to denote conjunctions of LMIs, and C, C_1, $C_2, \ldots$, A, A_1, $A_2, \ldots$ to denote conjunctions of LMCs. For a linear term t, we use $-t$ to denote the additive inverse of t modulo 2^p.

Proposition 1. *$(t_1 < t_2)$ is equivalent to both $(t_1 \leq 2^p - 2) \wedge (t_1 + 1 \leq t_2)$ and $(t_2 \geq 1) \wedge (t_1 \leq t_2 - 1)$.*

Proof of Proposition 1. $(t_1 < t_2)$ is equivalent to $((t_1 = 2^p - 1) \wedge (t_1 < t_2)) \vee ((t_1 \leq 2^p - 2) \wedge (t_1 < t_2))$. Note that $(t_1 = 2^p - 1) \wedge (t_1 < t_2)$ is equivalent to false. Moreover, $(t_1 \leq 2^p - 2) \wedge (t_1 < t_2)$ is equivalent to $(t_1 \leq 2^p - 2) \wedge (t_1 + 1 \leq t_2)$. Hence $(t_1 < t_2)$ is equivalent to $(t_1 \leq 2^p - 2) \wedge (t_1 + 1 \leq t_2)$.

$(t_1 < t_2)$ is equivalent to $((t_2 = 0) \wedge (t_1 < t_2)) \vee ((t_2 \geq 1) \wedge (t_1 < t_2))$. Since $(t_2 = 0) \wedge (t_1 < t_2)$ is equivalent to false and $(t_2 \geq 1) \wedge (t_1 < t_2)$ is equivalent to $(t_2 \geq 1) \wedge (t_1 \leq t_2 - 1)$, $(t_1 < t_2)$ is equivalent to $(t_2 \geq 1) \wedge (t_1 \leq t_2 - 1)$. □

Proposition 1 implies that there is no loss of generality in assuming that LMIs are restricted to be of the form $t_1 \leq t_2$. However, for clarity of exposition, we

allow LMIs of the form $t_1 < t_2$, whenever convenient.

Proposition 2. *An LME or LMD $t_1 \bowtie t_2$, where $\bowtie \in \{=, \neq\}$, can be equivalently expressed as $2^\mu \cdot x \bowtie t$, where t is a linear term free of x, and μ is an integer such that $0 \leq \mu \leq p$.*

Example of Proposition 2: All LMCs in this example have modulus 8. Consider the LME $7x + 4y = x + z$. Rearranging the terms modulo 8, we get $6x = 4y + z$, which can be written as $2^1 \cdot 3x = 4y + z$. Multiplying by 3 (multiplicative inverse of 3 modulo 8) and simplifying gives, $2^1 x = 4y + 3z$.

Proof of Proposition 2. Consider an LME $t_1 = t_2$. The linear term t_1 can be expressed as $a_1 \cdot x + s_1$, where a_1 is a constant and s_1 is a linear term free of x. Similarly, the linear term t_2 can be expressed as $a_2 \cdot x + s_2$, where a_2 is a constant and s_2 is a linear term free of x. Thus the LME $t_1 = t_2$ can be expressed as $(a_1 \cdot x + s_1 = a_2 \cdot x + s_2)$. Let a be $a_1 - a_2$ and s be $s_2 - s_1$. The LME can be expressed as $a \cdot x = s$.

If $a \neq 0$, then a can be expressed as $2^\mu \cdot b$, where μ is an integer such that $0 \leq \mu \leq p-1$ and b is an odd number. Thus we have $2^\mu \cdot b \cdot x = s$. Since b is odd, it has a multiplicative inverse modulo 2^p, say b'. Multiplying both sides of $2^\mu \cdot b \cdot x = s$ by b', we get the LME $2^\mu \cdot x = s \cdot b'$, which is in the required form since $s \cdot b'$ is free of x.

If $a = 0$, then a can be expressed as 2^p, and thus the LME $a \cdot x = s$ can be equivalently expressed as $2^p \cdot x = s$, where s is free of x.

Since an LMD $t_1 \neq t_2$ is equivalent to the negation of the LME $t_1 = t_2$, it is easy to see that it can be equivalently expressed as $2^\mu \cdot x \neq t$, where t is a linear term free of x, and μ is an integer such that $0 \leq \mu \leq p$. □

Definition of κ: For every linear term t_1 and variable x, we define $\kappa(x,t_1)$ to be an integer in $\{0,\ldots,p\}$ such that t_1 is equivalent to $2^{\kappa(x,t_1)}\cdot b\cdot x+t$, where t is a linear term free of x, and b is an odd number. Note that if t_1 is free of x, then $\kappa(x,t_1)=p$. The definition of $\kappa(x,\cdot)$ can be extended to (conjunctions of) LMCs as follows. Let c be an LME/LMD equivalent to $2^{\mu}\cdot x\bowtie t$, where $\bowtie\in\{=,\neq\}$ and t is free of x. We define $\kappa(x,c)$ to be μ in this case. If t_1,t_2 are linear terms, then $\kappa(x,t_1\leq t_2)$ is defined to be $min(\kappa(x,t_1),\kappa(x,t_2))$. Finally, if $c_1,\ldots,c_m$ are LMCs, then $\kappa(x,\bigwedge_{i=1}^{m}(c_i))$ is defined to be $\min_{i=1}^{m}(\kappa(x,c_i))$. Observe that if C is a conjunction of (possibly one) LMCs and if $\kappa(x,C)=k$, then only the least significant $p-k$ bits of x affect the satisfaction of C. We will say that x is in the support of C if $\kappa(x,C)<p$.

3.2 Layer1: Simplifications using LMEs

Layer1 involves simplification of the given conjunction of LMCs using the LMEs present in the conjunction. It is an extension of the work by Ganesh et. al. in [79]. The following Proposition and Lemmas form the crux of Layer1.

Proposition 3. *Let c be an LME $2^k\cdot x=t$, where k denotes $\kappa(x,c)$. Then $\exists x.\,c\equiv(2^{p-k}\cdot t=0)$.*

Example of Proposition 3: All LMCs in this example have modulus 8. $\exists x.\,(2^1x=5y+2)\equiv(2^{3-1}(5y+2)=0)\equiv(4y=0)$.

Proof of Proposition 3. Let φ_1 and φ_2 denote the formulas $\exists x.\,(2^k\cdot x=t)$ and $2^{p-k}\cdot t=0$ respectively. To see that $\varphi_1\Rightarrow\varphi_2$, we simply multiply both sides of $2^k\cdot x=t$ by 2^{p-k}, and simplify modulo 2^p. To see why $\varphi_2\Rightarrow\varphi_1$, note that φ_2

implies that the least significant k bits of t evaluate to zero. Also recall that t is free of x. Given any value of variables in t such that the least significant k bits of t evaluate to zero, we can always find a value of x such that $2^k \cdot x = t$. This can be done by choosing the least significant $p-k$ bits of x to be the same as the most significant $p-k$ bits of t. Hence, $\varphi_2 \Rightarrow \varphi_1$, and therefore $\varphi_1 \equiv \varphi_2$. □

Lemma 1. *Let A be a conjunction of LMEs. Then $\exists x.A$ can be equivalently expressed as a conjunction of LMEs each of which is free of x.*

Example of Lemma 1: All LMCs in this example have modulus 8. Consider the problem of computing $\exists x.((2^1x = 5y+2) \wedge (2^2x = 5y+6z) \wedge (2^1x = 2y+4))$. This can be equivalently expressed as $\exists x.((2x = 5y+2) \wedge (2\cdot(5y+2) = 5y+6z) \wedge (5y+2 = 2y+4))$. Simplifying modulo 8, we get $\exists x.((2x = 5y+2)) \wedge (5y+2z = 4) \wedge (3y = 2)$. Using Proposition 3, we obtain the final result as $(4y = 0) \wedge (5y+2z = 4) \wedge (3y = 2)$.

Proof of Lemma 1. Let A be $\bigwedge_{i=1}^{m}(q_i)$, where each q_i is an LME. Let each LME q_i be of the form $2^{k_i} \cdot x = t_i$, where $k_i = \kappa(x, q_i)$ and $1 \leq i \leq m$. Without loss of generality, let k_1 be the minimum of $k_1, \ldots, k_m$. It can be observed that the LME $2^{k_1} \cdot x = t_1$ can be used to eliminate the occurrences of x in other LMEs by expressing each LME $2^{k_i} \cdot x = t_i$ for $2 \leq i \leq m$ as $2^{\mu_i} \cdot t_1 = t_i$, where each $\mu_i = k_i - k_1$. Hence, $\exists x.A$ can be equivalently expressed as $C_1 \wedge \exists x.(2^{k_1} \cdot x = t_1)$, where C_1 is the conjunction of the LMEs $2^{\mu_i} \cdot t_1 = t_i$. Using Proposition 3, it follows that $C_1 \wedge \exists x.(2^{k_1} \cdot x = t_1)$ is equivalent to $C_1 \wedge (2^{p-k_1} \cdot t_1 = 0)$. □

Lemma 2. *Let A be a conjunction of LMCs containing at least one LME. Let $2^{k_1} \cdot x = t_1$ be the LME with the minimum $\kappa(x,\cdot)$ value among the LMEs in A. Then $\exists x.A \equiv C_1 \wedge \exists x.C_2$, where C_1 is a conjunction of LMCs free of x, and C_2 is a*

conjunction of $2^{k_1} \cdot x = t_1$ *and (possibly zero) LMIs and LMDs, each of which has* $\kappa(x, \cdot)$ *less than* k_1.

Example of Lemma 2: All LMCs in this example have modulus 8. Consider the problem of computing $\exists x.((2^1x = 5y+2) \wedge (2^0x \neq 6y+7z) \wedge (2^0 \cdot 5x+z \leq 2^1x) \wedge (2^1 \cdot 3x \leq y+z))$. Substituting the occurrences of 2^1x in the LMIs $(2^0 \cdot 5x+z \leq 2^1x)$ and $(2^1 \cdot 3x \leq y+z)$ by $5y+2$, we have $\exists x.((2x = 5y+2) \wedge (x \neq 6y+7z) \wedge (5x+z \leq 5y+2) \wedge (3 \cdot (5y+2) \leq y+z))$. Simplifying modulo 8, we get $(7y+6 \leq y+z) \wedge \exists x.((2x = 5y+2) \wedge (x \neq 6y+7z) \wedge (5x+z \leq 5y+2))$. Note that the result is of the form $C_1 \wedge \exists x.C_2$, as specified in Lemma 2.

Proof of Lemma 2. Let A be equivalent to $E \wedge D \wedge I$, where E is a conjunction of LMEs, D is a conjunction of LMDs, and I is a conjunction of LMIs. Let E be $\bigwedge_{i=1}^{m}(q_i)$, where each q_i is an LME, D be $\bigwedge_{i=m+1}^{n}(d_i)$, where each d_i is an LMD, and I be $\bigwedge_{i=n+1}^{r}(l_i)$, where each l_i is an LMI.

Suppose each LME q_i is of the form $2^{k_i} \cdot x = t_i$, where $k_i = \kappa(x, q_i)$ and $1 \leq i \leq m$. Suppose each LMD d_i is of the form $2^{k_i} \cdot x \neq t_i$, where $k_i = \kappa(x, d_i)$ and $m+1 \leq i \leq n$. In addition, suppose each LMI l_i is of the form $(a_i \cdot x+u_i \leq b_i \cdot x+v_i)$, where a_i, b_i constants such that $(a_i \neq 0) \vee (b_i \neq 0)$, u_i, v_i are linear terms free of x, and $n+1 \leq i \leq r$. Let us express each $a_i \cdot x$ appearing in the LMIs such that $a_i \neq 0$ in the equivalent form $2^{k_i} \cdot e_i \cdot x$, where $k_i = \kappa(x, a_i \cdot x)$ and e_i is an odd number. Similarly, let us express each $b_i \cdot x$ appearing in the LMIs such that $b_i \neq 0$ in the equivalent form $2^{k'_i} \cdot e'_i \cdot x$, where $k'_i = \kappa(x, b_i \cdot x)$ and e'_i is an odd number.

Without loss of generality, let k_1 be the minimum of $k_1, \ldots, k_m$. It can be observed that the LME $2^{k_1} \cdot x = t_1$ can be used to eliminate the occurrences of x in other LMEs, and in the LMDs and the LMIs with $\kappa(x, .)$ at least as large as k_1 in

the following way.

- Each LME $2^{k_i} \cdot x = t_i$ for $2 \leq i \leq m$ can be equivalently expressed as $2^{\mu_i} \cdot t_1 = t_i$ where each $\mu_i = k_i - k_1$.

- Each LMD $2^{k_i} \cdot x \neq t_i$ for $m+1 \leq i \leq n$, such that $k_1 \leq k_i$ can be equivalently expressed as $2^{\mu_i} \cdot t_1 \neq t_i$ where each $\mu_i = k_i - k_1$.

- Each occurrence of x of the form $2^{k_i} \cdot e_i \cdot x$ in the LMIs for $n+1 \leq i \leq r$ such that $k_1 \leq k_i$ can be equivalently expressed as $2^{\mu_i} \cdot t_1 \cdot e_i$ where each $\mu_i = k_i - k_1$.

- Each occurrence of x of the form $2^{k'_i} \cdot e'_i \cdot x$ in the LMIs for $n+1 \leq i \leq r$ such that $k_1 \leq k'_i$ can be equivalently expressed as $2^{\mu'_i} \cdot t_1 \cdot e'_i$ where each $\mu'_i = k'_i - k_1$.

Hence, it can be observed that $\exists x.A$ can be equivalently expressed as $C_1 \wedge \exists x.C_2$, where C_1 is a conjunction of LMCs free of x, and C_2 is a conjunction of the LME $2^{k_1} \cdot x = t_1$ and those LMIs and LMDs from A with $\kappa(x,.)$ less than k_1, after substitution of the occurrences of $2^{k_1} \cdot x$ by t_1. □

For the remainder of the chapter, we adopt the convention that algorithms for eliminating a single variable will have names starting with "*QE1_*",while those for eliminating multiple variables will have names starting with "*QE_*".

Proposition 3, Lemma 1, and Lemma 2 yield us a simple heuristic *QE1_Layer1* that forms the core of Layer1 of our QE algorithm. Given a conjunction of LMCs A and a variable x to be quantified, *QE1_Layer1* computes $\exists x.A$ as $C_1 \wedge \exists x.C_2$ based on Lemma 2. If the $\kappa(x,\cdot)$ of all LMDs and LMIs in A are at least as large as k_1 (as in Lemma 2), then C_2 consists of the single LME $2^{k_1} \cdot x = t_1$. In this case,

$\exists x.C_2$ simplifies to $2^{p-k_1} \cdot t_1 = 0$ (see Proposition 3), and *QE1_Layer1* suffices to compute $\exists x.A$. However, in general, C_2 may contain LMDs and LMIs with $\kappa(x,\cdot)$ values less than k_1. We describe techniques to address such cases in the following sections.

***Analysis of Complexity*:** Consider a conjunction of LMCs with a subset of variables in its support to be eliminated. Let n be the number of LMCs in the conjunction, v be the number of variables its support, and e be the number of variables to be eliminated. It can be observed that for a variable x to be eliminated, Layer1 performs $O(n \cdot v)$ multiplications and additions in the worst-case. Assuming that arithmetic operations on p-bit numbers take time $O(Q(p))$ in the worst-case, where $Q(p)$ is a polynomial on p such that $p \leq Q(p) \leq p^3$, elimination of a variable hence has a worst-case time complexity of $O(n \cdot v \cdot Q(p))$. Observe that eliminating a variable does not increase the number of LMCs in the conjunction. Hence eliminating e variables has a worst-case time complexity of $O(e \cdot n \cdot v \cdot Q(p))$. Note that reading and writing an LMC with v variables in support takes $O(v \cdot p)$ time. Hence reading n LMCs as input and writing them back after eliminating the variables takes $O(n \cdot v \cdot p)$ time. Hence Layer1 has a worst-case time complexity of $O(e \cdot n \cdot v \cdot Q(p) + n \cdot v \cdot p)$. Since $p \leq Q(p) \leq p^3$, this reduces to $O(e \cdot n \cdot v \cdot Q(p)$.

3.3 Layer2: Dropping Unconstraining LMIs and LMDs

Formally, our goal in this section is to express C_2, obtained after application of *QE1_Layer1*, as $C \wedge D \wedge I$, where (i) D is a conjunction of (zero or more) LMDs in C_2, (ii) I is a conjunction of (zero or more) LMIs in C_2, (iii) C is the conjunction of

the remaining LMCs in C_2, and (iv) $\exists x.(C) \Rightarrow \exists x.(C \wedge D \wedge I)$. Since $\exists x.(C \wedge D \wedge I) \Rightarrow \exists x.(C)$ always holds, this would allow us to compute $\exists x.C_2$, or equivalently $\exists x.(C \wedge D \wedge I)$, as $\exists x.C$. We say that D and I are *unconstraining* LMDs and LMIs, respectively, in such cases.

Given C, D and I satisfying conditions (i), (ii) and (iii) above, checking if condition (iv) holds requires solving a quantified bit-vector formula in general. This can be done by using an SMT solver such as Z3 that supports quantified bit-vector formulae. Alternatively bit-blasting followed by QBF solving or bit-level QE can be used. However applying such techniques can be expensive, as demonstrated in our experiments. In the following discussion, we focus on finding sufficient and polynomial time computable conditions for condition (iv) to hold.

Let $x[i]$ denote the i^{th} bit of a bit-vector x, where $x[0]$ denotes its least significant bit. For $i \leq j$, let $x[i:j]$ denote the slice of bit-vector x consisting of bits $x[i]$ through $x[j]$. Given slice $x[i:j]$, its value is the natural number encoded by the bits in the slice. A key notion used in the subsequent discussion is that of "adapting" a solution of a constraint to make it satisfy another constraint. Formally, we say that a solution θ_1 of a conjunction φ of LMCs can be adapted with respect to slice $x[i:j]$ to satisfy a (possibly different) conjunction ψ of LMCs if there exists a solution θ_2 of ψ that matches θ_1 except possibly in the bits of slice $x[i:j]$.

Example: Consider the LMCs $(x = y + z) \pmod 8$ and $(4y + z \leq x) \pmod 8$. Let θ_1 be the solution $x = 1$, $y = 1$, $z = 0$ of $(x = y + z) \pmod 8$, and let θ_2 be the solution $x = 5$, $y = 1$, $z = 0$ of $(4y + z \leq x) \pmod 8$. Note that θ_2 matches θ_1 except in the bits of slice $x[2:2]$. Hence we can say that θ_1 can be adapted with respect to slice $x[2:2]$ to satisfy $(4y + z \leq x) \pmod 8$.

The central idea in the second layer of our QE algorithm is to efficiently com-

pute an under-approximation η of the number of ways in which an *arbitrary* solution of C can be adapted to satisfy $C \wedge D \wedge I$. It is easy to see that if $\eta \geq 1$, then $\exists x.\,(C) \Rightarrow \exists x.\,(C \wedge D \wedge I)$. We illustrate this idea below through an example. We will use this as a running example throughout this section.

Example: Consider the problem of computing $\exists x.\,(C \wedge D \wedge I)$, where $C \equiv (z = 4x + y)$, $D \equiv (x \neq z + 7)$, and $I \equiv (6x + y \leq 4)$ and all LMCs have modulus 8. We claim that an arbitrary solution of C can be adapted to satisfy $C \wedge D \wedge I$. Note that C constrains only slice $x[0:0]$, whereas I constraints slice $x[0:1]$ and D constraints slice $x[0:2]$. Therefore, the value of slice $x[1:2]$ does not affect satisfaction of C, and the value of slice $x[2:2]$ does not affect satisfaction of $C \wedge I$. *Any solution* of C can be adapted with respect to slice $x[1:1]$ to satisfy I by choosing value of slice $x[1:1]$ such that $6x$ lies between $-y$ and $4-y$. Since $x[0:0]$ is unchanged, each such adapted solution must also satisfy $C \wedge I$. For example, the solution $x = 1$, $y = 0$, $z = 4$ of C can be adapted with respect to slice $x[1:1]$ to obtain the solution $x = 3$, $y = 0$, $z = 4$ of $C \wedge I$. Moreover, *any solution* of $C \wedge I$ can be adapted with respect to slice $x[2:2]$ to satisfy D by choosing value for slice $x[2:2]$ that differs from the most significant bit of $z + 7$. Since $x[0:1]$ is unchanged, each such adapted solution also satisfies $C \wedge D \wedge I$. For example, the solution $x = 3$, $y = 0$, $z = 4$ of $C \wedge I$ can be adapted with respect to slice $x[2:2]$ to obtain the solution $x = 7$, $y = 0$, $z = 4$ of $C \wedge D \wedge I$. In this case, Layer2 computes the under-approximation η of the number of ways in which an arbitrary solution of C can be adapted to satisfy $C \wedge D \wedge I$ as ≥ 1, thus inferring that $\exists x.\,(C) \Rightarrow \exists x.\,(C \wedge D \wedge I)$.

Our technique of dropping unconstraining LMCs is conceptually similar to clause-elimination procedures used in SAT solvers. Given a propositional formula in CNF, clause-elimination procedures identify redundant clauses and drop

them without changing the satisfiability or unsatisfiability of the formula. The works in [70, 71, 72] focus on different kinds of redundant clauses such as tautological clauses, subsumed clauses, blocked clauses, and covered clauses, and present procedures to eliminate them. The recent work by Kiesl et. al in [73] presents more generalized redundant clauses called set-blocked clauses and super-blocked clauses, and gives detailed complexity analysis of the problem of eliminating them.

We now present procedure *QE1_Layer2*, that applies the technique described above to problem instances of the form $\exists x.C_2$, obtained after invoking *QE1_Layer1*. *QE1_Layer2* initially expresses $\exists x.C_2$ as $\exists x.(C \wedge D \wedge I)$, where C denotes $2^{k_1} \cdot x = t_1$ and $D \wedge I$ denotes the conjunction of LMDs and LMIs in C_2. If η (as defined above) is at least 1, then $D \wedge I$ is dropped from C_2. Otherwise, the LMCs in $D \wedge I$ with the largest $\kappa(x,\cdot)$ value (i.e. LMCs whose satisfaction depends on the least number of bits of x) are identified and included in C, and the above process repeats. If all the LMIs and LMDs in $\exists x.C_2$ are dropped in this manner, then $\exists x.C_2$ reduces to $\exists x.(2^{k_1} \cdot x = t_1)$, and *QE1_Layer2* can return the equivalent form $2^{p-k_1} \cdot t_1 = 0$. Otherwise, *QE1_Layer2* returns $\exists x.C_3$, where C_3 is a conjunction of possibly fewer LMCs compared to C_2, such that $\exists x.C_3 \equiv \exists x.C_2$.

Before presenting the details of computing η, we present the following proposition.

Proposition 4. *Let $x_1, \ldots, x_n$ be r-bit numbers and b be an r-bit odd number such that $b \cdot x_1, \ldots, b \cdot x_n$ take distinct consecutive values. Let ℓ be a number such that $1 \leq \ell \leq r$. If $n < 2^\ell$, then the values of $x_1[0:\ell-1], \ldots, x_n[0:\ell-1]$ are distinct. Otherwise, if $n \geq 2^\ell$, then the values of $x_1[0:\ell-1], \ldots, x_n[0:\ell-1]$ span the entire range $0, 1, \ldots, 2^\ell - 1$.*

Example of Proposition 4: Suppose $r = 3$, $b = 3$, and $n = 5$. Let $b \cdot x_1$, $b \cdot x_2$, $b \cdot x_3$, $b \cdot x_4$, $b \cdot x_5$ be $2, 3, 4, 5, 6$. By multiplying by the multiplicative inverse of b modulo 2^r, i.e., 3, we get the corresponding values of x_1, x_2, x_3, x_4, x_5 as 6, 1, 4, 7, 2.

- Case 1: Let ℓ be 3. Hence $n < 2^\ell$. The values of $x_1[0 : \ell - 1]$, $x_2[0 : \ell - 1]$, $x_3[0 : \ell - 1]$, $x_4[0 : \ell - 1]$, $x_5[0 : \ell - 1]$ are 6, 1, 4, 7, 2, which are distinct.
- Case 2: Let ℓ be 2. Hence $n \geq 2^\ell$. The values of $x_1[0 : \ell - 1]$, $x_2[0 : \ell - 1]$, $x_3[0 : \ell - 1]$, $x_4[0 : \ell - 1]$, $x_5[0 : \ell - 1]$ are 2, 1, 0, 3, 2, which span the entire range $0, 1, \ldots, 2^\ell - 1$.

Proof of Proposition 4. The proof is based on the following observations:

1. The values of $(b \cdot x_1)[0 : \ell - 1], \ldots, (b \cdot x_n)[0 : \ell - 1]$ are consecutive.
2. $(b \cdot x_i)[0 : \ell - 1]$ is equivalent to $b[0 : \ell - 1] \cdot x_i[0 : \ell - 1]$ for $1 \leq i \leq n$.
3. $b[0 : \ell - 1]$ is odd.

Since $b[0 : \ell - 1]$ is odd, it has a multiplicative inverse $(b[0 : \ell - 1])'$ modulo 2^ℓ. Note that $(b[0 : \ell - 1])'$ is also odd modulo 2^ℓ. Since $(b \cdot x_i)[0 : \ell - 1]$ is equivalent to $b[0 : \ell - 1] \cdot x_i[0 : \ell - 1]$ for $1 \leq i \leq n$, we get values of $x_1[0 : \ell - 1], \ldots, x_n[0 : \ell - 1]$ by multiplying the values of $(b \cdot x_1)[0 : \ell - 1], \ldots, (b \cdot x_n)[0 : \ell - 1]$ by $(b[0 : \ell - 1])'$ modulo 2^ℓ.

Observe that for $1 \leq i \leq n$ and $1 \leq j \leq n$ such that $i \neq j$, $x_i[0 : \ell - 1] = x_j[0 : \ell - 1]$ iff $(b \cdot x_i)[0 : \ell - 1] = (b \cdot x_j)[0 : \ell - 1]$. Since the values of $(b \cdot x_1)[0 : \ell - 1]$, $\ldots, (b \cdot x_n)[0 : \ell - 1]$ are consecutive, it follows that, if $n < 2^\ell$, then the values of $x_1[0 : \ell - 1], \ldots, x_n[0 : \ell - 1]$ are distinct. If $n \geq 2^\ell$, then the values of $(b \cdot x_1)[0 :$

$\ell-1], \ldots, (b \cdot x_n)[0:\ell-1]$ are consecutive and they span the range $0, 1, \ldots, 2^\ell - 1$. Hence it is obvious that the values of $x_1[0:\ell-1]$, $\ldots$,$x_n[0:\ell-1]$ also span the range $0, 1, \ldots, 2^\ell - 1$. □

Computing η: Let I be $\bigwedge_{i=1}^{n}(l_i)$, where each l_i is an LMI of the form $s_i \bowtie t_i$, the operator $\bowtie$ is in $\{\leq, \geq\}$, s_i is a linear term with x in its support, and t_i is a linear term free of x. Note that this implies some loss of generality, since we disallow LMIs of the form $s \bowtie t$, where both s and t have x in their support. However, our experiments indicate that this is not very restrictive in practice. Let $s_1, \ldots, s_r$ be the distinct linear terms in I with x in their support. We partition I into $I_1, \ldots, I_r$, where each I_j is the conjunction of only those LMIs in I that contain the linear term s_j. We assume without loss of generality that each I_j contains the trivial LMIs $s_j \geq 0$ and $s_j \leq 2^p - 1$. Let I_j have n_j LMIs, of which the first $m_j (< n_j)$ are of the form $s_j \geq t_q$, where $1 \leq q \leq m_j$. Let the remaining LMIs in I_j be of the form $s_j \leq t_q$, where $m_j + 1 \leq q \leq n_j$.

Consider the inequality $Z_j : u_j \leq s_j \leq v_j$, where u_j denotes $\max_{q=1}^{m_j}(t_q)$ and v_j denotes $\min_{q=m_j+1}^{n_j}(t_q)$. Although Z_j is not a LMI, it is semantically equivalent to I_j. For notational convenience, let us denote $\kappa(x, s_j)$ by k_j. Clearly, the value of slice $x[p-k_j : p-1]$ does not affect the satisfaction of Z_j. We wish to compute the number of ways, say N_j, in which an arbitrary solution of C can be adapted with respect to slice $x[0 : p-k_j-1]$ to satisfy Z_j. Towards this end, we compute an integer δ_j in $\{0, \ldots, 2^p - 1\}$ such that $\delta_j \leq \max(v_j - u_j + 1, 0)$ for every combination of values of other variables. Intuitively, δ_j represents the minimum number of *consecutive* values that s_j can take for every combination of values of other variables, if we were to treat s_j as a fresh p-bit variable and if Z_j were to be satisfied.

Example: In our running example, where $C \equiv (z = 4x + y)$, $D \equiv (x \neq z + 7)$, and $I \equiv (6x + y \leq 4)$, we have $s_1 = 6x + y$ and $I_1 \equiv (6x + y \geq 0) \wedge (6x + y \leq 4) \wedge (6x + y \leq 7)$. Hence Z_1 is $(0 \leq 6x + y \leq 4)$ and thus $u_1 = 0$ and $v_1 = 4$. Note that $p = 3$, $k_1 = 1$, and the value of slice $x[2:2]$ does not affect the satisfaction of $(0 \leq 6x + y \leq 4)$. We are trying to compute N_1, the number of ways in which an arbitrary solution of $(z = 4x + y)$ can be adapted with respect to slice $x[0:1]$ to satisfy $(0 \leq 6x + y \leq 4)$. Treating $6x + y$ as a fresh variable f gives us $(0 \leq f \leq 4)$. As f can take five *consecutive* values in $(0 \leq f \leq 4)$, δ_1 is 5.

Lemma 3. *For every combination of values of variables other than x, there exist at least* $\lfloor \delta_j / 2^{k_j} \rfloor$ *distinct values that* $x[0 : p - k_j - 1]$ *can take while satisfying* Z_j.

Example of Lemma 3: In our example, $Z_1 \equiv (0 \leq 6x + y \leq 4)$, $p = 3$, $k_1 = 1$ and $\delta_1 = 5$. Note that, for every value of y, there are at least $\lfloor \delta_1 / 2^{k_1} \rfloor = \lfloor 5/2^1 \rfloor = 2$ distinct values that $x[0:1]$ can take while satisfying $(0 \leq 6x + y \leq 4)$.

Proof of Lemma 3. δ_j is the minimum number of *consecutive* values that s_j can take for every combination of values of other variables, if we were to treat s_j as a fresh p-bit variable and if $Z_j : u_j \leq s_j \leq v_j$ were to be satisfied. However, in general, s_j is of the form $2^{k_j} \cdot b_j \cdot x + w_j$, where w_j is a linear term free of x, and b_j is an odd number. Therefore, for every combination of values of variables other than x, there exist at least $\lfloor \delta_j / 2^{k_j} \rfloor$ *consecutive* values that $b_j[0 : p - k_j - 1] \cdot x[0 : p - k_j - 1]$ can take while satisfying Z_j. Since b_j is odd, $b_j[0 : p - k_j - 1]$ is odd. Let us apply Proposition 4 on these *consecutive* values of $b_j[0 : p - k_j - 1] \cdot x[0 : p - k_j - 1]$ with $n = \lfloor \delta_j / 2^{k_j} \rfloor$, $r = \ell = p - k_j$ and $b = b_j[0 : p - k_j - 1]$. Note that $\lfloor \delta_j / 2^{k_j} \rfloor < 2^{p - k_j}$, since $\delta_j < 2^p$. Therefore, using Proposition 4, we have: for every combination of values of variables other than x, there exist at least $\lfloor \delta_j / 2^{k_j} \rfloor$

distinct values that $x[0 : p-k_j-1]$ can take while satisfying Z_j. □

Lemma 3 indicates that there are at least $\lfloor \delta_j/2^{k_j} \rfloor$ ways in which an arbitrary solution of C can be adapted with respect to slice $x[0 : p-k_j-1]$ to satisfy Z_j. Hence, $N_j \geq \lfloor \delta_j/2^{k_j} \rfloor$. For notational convenience, we denote $\lfloor \delta_j/2^{k_j} \rfloor$ by $\widehat{N_j}$.

To understand how δ_j is computed in general, recall that for every g in $\{1 \ldots m_j\}$ and for every h in $\{m_j+1 \ldots n_j\}$, we have $t_g \leq s_j \leq t_h$. For every such pair of indices g and h, let $\delta_{g,h}$ be an integer in $\{0, \ldots, 2^p-1\}$ such that $\delta_{g,h} \leq \max(t_h - t_g + 1, 0)$ for every combination of values of t_h and t_g. The value of δ_j can then be obtained as the minimum of all $\delta_{g,h}$ values. For reasons of simplicity and efficiency, we compute the values of $\delta_{g,h}$ conservatively using the following Proposition.

Proposition 5. *1. If t_g and t_h are constants and $t_h \geq t_g$, then $\delta_{g,h} = t_h - t_g + 1$.*

2. If t_h is a constant, t_g can be expressed as $2^\tau \cdot t$, where τ is an integer such that $0 \leq \tau \leq p-1$, and $t_h \geq 2^p - 2^\tau$, then $\delta_{g,h} = t_h - (2^p - 2^\tau) + 1$.

3. If t_g is a constant, t_h can be expressed as $2^\tau \cdot t + a$, where τ is an integer such that $0 \leq \tau \leq p-1$, and $a \bmod 2^\tau \geq t_g$, then $\delta_{g,h} = a \bmod 2^\tau - t_g + 1$.

4. Otherwise $\delta_{g,h} = 0$.

Example of Proposition 5:

1. Suppose $t_g = 1$ and $t_h = 6$. Therefore, $\max(t_h - t_g + 1, 0) = t_h - t_g + 1 = 6$. Since $\delta_{g,h} \leq \max(t_h - t_g + 1, 0)$, we can set $\delta_{g,h}$ to 6.

2. Suppose $t_g = 4y$, $t_h = 14$, and $p = 4$. Here t_g is of the form $2^\tau \cdot t$, where $\tau = 2$ and $t = y$. Observe that the maximum possible value of $4y$ with modulus 16

is $2^p - 2^\tau = 12$, i.e., $4y \leq 12$. Therefore, $\max(t_h - t_g + 1, 0) = \max(14 - 4y + 1, 0) \geq 14 - 12 + 1 = 3$. Hence 3 can be used as $\delta_{g,h}$.

3. Suppose $t_g = 0$, $t_h = 4y + 7$, and $p = 4$. Here t_h is of the form $2^\tau \cdot t + a$, where $\tau = 2$, $t = y$, and $a = 7$. Observe that the minimum possible value of $4y + 7$ with modulus 16 is $a \bmod 2^\tau = 7 \bmod 4 = 3$, i.e., $4y + 7 \geq 3$. Therefore, $\max(t_h - t_g + 1, 0) = \max(4y + 7 - 0 + 1, 0) \geq 3 - 0 + 1 = 4$. Hence 4 can be used as $\delta_{g,h}$.

4. Suppose $t_g = y$, $t_h = z$. In such cases we set $\delta_{g,h}$ to 0.

Proof of Proposition 5. $\delta_{g,h}$ is an integer in $\{0, \ldots, 2^p - 1\}$ such that $\delta_{g,h} \leq \max(t_h - t_g + 1, 0)$ for every combination of values of t_h and t_g.

1. If t_g and t_h are constants and $t_h \geq t_g$, then $\max(t_h - t_g + 1, 0)$ reduces to $t_h - t_g + 1$. Therefore, it is obvious that $t_h - t_g + 1$ can be used as $\delta_{g,h}$.

2. Consider the case when t_h is a constant, t_g can be expressed as $2^\tau \cdot t$, where τ is an integer such that $0 \leq \tau \leq p - 1$, and $t_h \geq 2^p - 2^\tau$. Since t_g is a multiple of 2^τ, the possible values of t_g are $0, 2^\tau, \ldots, 2^p - 2^\tau$. Hence the maximum possible value of t_g is $2^p - 2^\tau$, i.e., $t_g \leq 2^p - 2^\tau$. This implies that $t_h - (2^p - 2^\tau) + 1 \leq \max(t_h - t_g + 1, 0)$. Hence $t_h - (2^p - 2^\tau) + 1$ can be used as $\delta_{g,h}$.

3. Consider the case when t_g is a constant, t_h can be expressed as $2^\tau \cdot t + a$, where τ is an integer such that $0 \leq \tau \leq p - 1$, and $a \bmod 2^\tau \geq t_g$. Let $a = 2^\tau \cdot a_1 + a_2$, where $a_2 = a \bmod 2^\tau$ and $a_1 \geq 0$. Hence t_h can be expressed as $2^\tau \cdot (t + a_1) + a_2$. Since $2^\tau \cdot (t + a_1)$ is a multiple of 2^τ, the possible values of $2^\tau \cdot (t + a_1)$ are $0, 2^\tau, \ldots, 2^p - 2^\tau$. Hence the possible values of t_h are

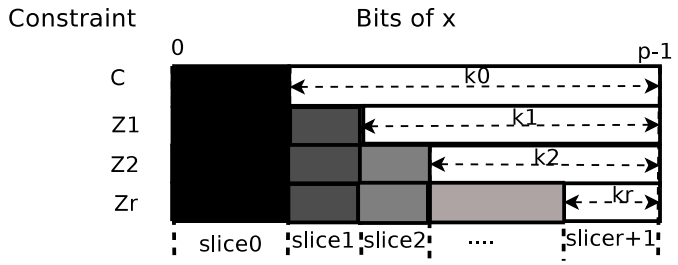

Figure 3.1: Slicing of bits of x by $k_0, \ldots, k_r$

$a_2, 2^\tau + a_2, \ldots, 2^p - 2^\tau + a_2$. Therefore, the minimum possible value of t_h is a_2, i.e., $t_h \geq a_2$, which implies that $a_2 - t_g + 1 \leq \max(t_h - t_g + 1, 0)$. Hence $a_2 - t_g + 1$, i.e., $a \bmod 2^\tau - t_g + 1$ can be used as $\delta_{g,h}$.

4. Consider the case when none of the above conditions is true. Since $0 \leq \max(t_h - t_g + 1, 0)$, we can use $\delta_{g,h}$ as 0 in this case.

□

Let D be $\bigwedge_{i=1}^{m}(d_i)$, where each d_i is an LMD of the form $2^{\kappa(x,d_i)} \cdot x \neq t_{d_i}$, where t_{d_i} is a linear term free of x. Let k_0 denote $\kappa(x,C)$, and let C be such that k_0 is greater than both $\max_{i=1}^{m} \kappa(x,d_i)$ and $\max_{j=1}^{r} k_j$ (recall that $k_j = \kappa(x,s_j)$). To simplify the exposition, suppose further that $k_1 > \ldots > k_r$. We partition the bits of x into $r+2$ slices as shown in Fig. 3.1, where slice_0 represents $x[0 : p - k_0 - 1]$, slice_j represents $x[p - k_{j-1} : p - k_j - 1]$ for $1 \leq j \leq r$, and slice_{r+1} represents $x[p - k_r : p - 1]$. Note that the value of slice_0 potentially affects the satisfaction of C as well as that of Z_1 through Z_r, the value of slice_j potentially affects the satisfaction of Z_j through Z_r for $1 \leq j \leq r$, and the value of slice_{r+1} does not affect the satisfaction of any Z_j or C.

Let Z_0 denote True. Let θ be a solution of $C \wedge Z_0 \wedge \ldots \wedge Z_i$, where $0 \leq i < r$.

Note that bits in slice_{i+1} through slice_{r+1} do not affect satisfaction of $C \wedge Z_0 \wedge \ldots \wedge Z_i$. Let $Y_{i,j}$ denote the number of ways in which θ can be adapted with respect to bits in slice_{i+1} through slice_j, to satisfy Z_j, where $i < j \leq r$. Since slice_0 through slice_i are unchanged, each such adapted solution must also satisfy $C \wedge Z_0 \wedge \ldots \wedge Z_i$.

Lemma 4. *An arbitrary solution of $C \wedge Z_0 \wedge \ldots \wedge Z_i$ for $0 \leq i < r$ can be adapted with respect to bits in slice_{i+1} through slice_j, to satisfy Z_j for $i < j \leq r$ in at least $\lfloor \widehat{N_j}/2^{p-k_i} \rfloor$ ways. Moreover, if we focus only on slice_{i+1}, then there are at least $\min(\lfloor \widehat{N_j}/2^{p-k_i} \rfloor, 2^{k_i-k_{i+1}})$ distinct values of slice_{i+1} in the corresponding adapted solutions.*

Example of Lemma 4: In our running example, since $p = 3$, $k_0 = 2$, $k_1 = 1$, the bits of x are partitioned into three slices: slice_0 is $x[0:0]$, slice_1 is $x[1:1]$ and slice_2 is $x[2:2]$. Clearly, the value of slice_0 potentially affects the satisfaction of $(z = 4x + y)$ as well as that of $(0 \leq 6x + y \leq 4)$. The value of slice_1 potentially affects the satisfaction of $(0 \leq 6x + y \leq 4)$, but not that of $(z = 4x + y)$, and the value of slice_2 does not affect the satisfaction of $(z = 4x + y)$ or $(0 \leq 6x + y \leq 4)$. Let θ be a solution of $(z = 4x + y)$. Using Lemma 4, there exists at least $\lfloor \widehat{N_1}/2^{p-k_1} \rfloor = \lfloor \widehat{2}/2^{3-2} \rfloor = 1$ way in which θ can be adapted with respect to bits in slice_1 to satisfy $(0 \leq 6x + y \leq 4)$. Since slice_0 is unchanged, the adapted solution must satisfy $(z = 4x + y) \wedge (0 \leq 6x + y \leq 4)$.

Proof of Lemma 4. Recall that for every combination of values of variables other than x, there exist at least $\widehat{N_j}$ *consecutive* values that $b_j[0 : p - k_j - 1] \cdot x[0 : p - k_j - 1]$ can take while satisfying Z_j, where $b_j[0 : p - k_j - 1]$ is odd modulo 2^{p-k_j}. Let us apply Proposition 4 on these *consecutive* values of $b_j[0 : p - k_j - 1] \cdot x[0 :$

$p-k_j-1]$ with $n=\widehat{N_j}$, $r=p-k_j$, $\ell=p-k_i$ and $b=b_j[0:p-k_j-1]$. Using Proposition 4, we have: for every combination of values of variables other than x, (i) if $\widehat{N_j}<2^{p-k_i}$, there exist at least $\widehat{N_j}$ *distinct* values that $x[0:p-k_i-1]$ can take while satisfying Z_j, and (ii) if $\widehat{N_j}\geq 2^{p-k_i}$, the values that $x[0:p-k_i-1]$ can take while satisfying Z_j span the entire range $0,1,\ldots,2^{p-k_i}-1$.

Using Lemma 3, we know that, for every combination of values of variables other than x, there exist at least $\widehat{N_j}$ *distinct* values that can be assigned to $x[0:p-k_j-1]$ (i.e. bits in slice_0 through $\mathsf{slice}_\mathsf{j}$) while satisfying Z_j. This implies that for every combination of values of variables other than x and for any arbitrary value of $x[0:p-k_i-1]$ (i.e. bits in slice_0 through $\mathsf{slice}_\mathsf{i}$), there exist at least $\lfloor\widehat{N_j}/2^{p-k_i}\rfloor$ *distinct* values that can be assigned to $x[p-k_i:p-k_j-1]$ (i.e. bits in $\mathsf{slice}_{\mathsf{i}+1}$ through $\mathsf{slice}_\mathsf{j}$) while satisfying Z_j. Hence, an arbitrary solution of $C\wedge Z_0\wedge\ldots\wedge Z_i$ for $0\leq i<r$ can be adapted with respect to bits in $\mathsf{slice}_{\mathsf{i}+1}$ through $\mathsf{slice}_\mathsf{j}$, to satisfy Z_j for $i<j\leq r$ in at least $\lfloor\widehat{N_j}/2^{p-k_i}\rfloor$ ways.

In order to prove our claim on values of $\mathsf{slice}_{\mathsf{i}+1}$ in the corresponding adapted solutions, again apply Proposition 4 on values of $b_j[0:p-k_j-1]\cdot x[0:p-k_j-1]$ with $n=\widehat{N_j}$, $r=p-k_j$, $\ell=p-k_{i+1}$ and $b=b_j[0:p-k_j-1]$. We have: for every combination of values of variables other than x, (i) if $\widehat{N_j}<2^{p-k_{i+1}}$, there exist at least $\widehat{N_j}$ *distinct* values that $x[0:p-k_{i+1}-1]$ can take while satisfying Z_j, and (ii) if $\widehat{N_j}\geq 2^{p-k_{i+1}}$, the values that $x[0:p-k_{i+1}-1]$ can take while satisfying Z_j span the entire range $0,1,\ldots,2^{p-k_{i+1}}-1$. In other words, for every combination of values of variables other than x, there exist at least $\min(\widehat{N_j},2^{p-k_{i+1}})$ *distinct* values that $x[0:p-k_{i+1}-1]$ can take while satisfying Z_j.

We have already seen that, for every combination of values of variables other than x, (i) if $\widehat{N_j}<2^{p-k_i}$, there exist at least $\widehat{N_j}$ *distinct* values that $x[0:p-k_i-1]$

can take while satisfying Z_j, and (ii) if $\widehat{N_j} \geq 2^{p-k_i}$, the values that $x[0:p-k_i-1]$ can take while satisfying Z_j span the entire range $0,1,\ldots,2^{p-k_i}-1$. This implies that, for every combination of values of variables other than x and for any arbitrary value of $x[0:p-k_i-1]$ (i.e. bits in slice_0 through $\mathsf{slice}_\mathsf{i}$), there exist at least $\min(\lfloor \widehat{N_j}/2^{p-k_i} \rfloor, \lfloor 2^{p-k_{i+1}}/2^{p-k_i} \rfloor) = \min(\lfloor \widehat{N_j}/2^{p-k_i} \rfloor, 2^{k_i-k_{i+1}})$ *distinct* values that can be assigned to $x[p-k_i:p-k_{i+1}-1]$ (i.e. bits in $\mathsf{slice}_{\mathsf{i}+1}$) while satisfying Z_j. Therefore, if we focus only on $\mathsf{slice}_{\mathsf{i}+1}$ in the aforementioned adapted solutions, then there are at least $\min(\lfloor \widehat{N_j}/2^{p-k_i} \rfloor, 2^{k_i-k_{i+1}})$ *distinct* values of $\mathsf{slice}_{\mathsf{i}+1}$. □

Using Lemma 4, we have $Y_{i,j} \geq \lfloor \widehat{N_j}/2^{p-k_i} \rfloor$. For notational convenience, let us denote $\min(\lfloor \widehat{N_j}/2^{p-k_i} \rfloor, 2^{k_i-k_{i+1}})$ by $\alpha_{i,j}$.

Lemma 4 indicates that a solution θ of $C \wedge Z_0 \wedge \ldots \wedge Z_i$ for $0 \leq i < r$ can be adapted to satisfy $C \wedge Z_0 \wedge \ldots \wedge Z_i \wedge Z_j$ for $i < j \leq r$ by using at least $\alpha_{i,j}$ different values of $\mathsf{slice}_{\mathsf{i}+1}$. Let the corresponding set of values of $\mathsf{slice}_{\mathsf{i}+1}$ be denoted $S^{\theta}_{i+1,j}$. If $\bigcap_{j=i+1}^{r} S^{\theta}_{i+1,j}$ is non-empty, there exists a common value of $\mathsf{slice}_{\mathsf{i}+1}$ that permits us to adapt θ with respect to $\mathsf{slice}_{\mathsf{i}+1}$ through $\mathsf{slice}_\mathsf{r}$ to satisfy Z_{i+1} through Z_r, respectively. It is therefore desirable to have $|\bigcap_{j=i+1}^{r} S^{\theta}_{i+1,j}| \geq 1$. Using the Inclusion-Exclusion principle, we find that $|\bigcap_{j=i+1}^{r} S^{\theta}_{i+1,j}| \geq (\sum_{j=i+1}^{r} \alpha_{i,j}) - (r-i-1) \cdot 2^{k_i-k_{i+1}}$. Note that the lower bound is independent of θ. For notational convenience, let us denote the lower bound by W_{i+1}.

If $W_{i+1} \geq 1$ for all $i \in \{0,\ldots r-1\}$, an arbitrary solution θ of C can be adapted to satisfy $C \wedge Z_0 \wedge \ldots \wedge Z_r$ as follows. Since $W_1 \geq 1$, we choose a value of slice_1, say v_1, from $\bigcap_{j=1}^{r} S^{\theta}_{1,j}$. Let θ_1 denote θ with slice_1 (possibly) changed to have value v_1. Then θ_1 satisfies $C \wedge Z_1$. Since $W_2 \geq 1$, we can now choose a value of slice_2, say v_2, from $\bigcap_{j=2}^{r} S^{\theta_1}_{2,j}$, and repeat the procedure until we have chosen values for slice_1 through $\mathsf{slice}_\mathsf{r}$. Finally, since $\mathsf{slice}_{\mathsf{r}+1}$ does not affect the satisfaction of

C or of any Z_i, we can choose an arbitrary value for slice_{r+1}. Clearly, there are at least $(\prod_{i=0}^{r-1}|W_{i+1}|)\cdot 2^{k_r}$ ways in which values of different slices can be chosen, so as to adapt θ to satisfy $C \wedge Z_0 \wedge \ldots \wedge Z_r$. Let us denote $(\prod_{i=0}^{r-1}|W_{i+1}|)\cdot 2^{k_r}$ by μ_I.

Example (Continued): We have $Y_{0,1} \geq \lfloor \widehat{N_1}/2^{p-k_0} \rfloor = 1$. Also $\alpha_{0,1} = \min(\lfloor \widehat{N_1} / 2^{p-k_0} \rfloor, 2^{k_0-k_1}) = \min(1, 2^{2-1}) = 1$. Hence $W_1 = (\sum_{j=1}^{1}\alpha_{0,1}) - (1-0-1)\cdot 2^{k_0-k_1} = \alpha_{0,1} = 1$. Note that there is at least one way of adapting an arbitrary solution of $(z = 4x+y)$ with respect to slice_1 to satisfy $(z = 4x+y) \wedge (0 \leq 6x+y \leq 4)$. Moreover, there are at least two ways of adapting an arbitrary solution of $(z = 4x+y)$ with respect to slice_1 through to slice_2 to satisfy $(z = 4x+y) \wedge (0 \leq 6x+y \leq 4)$ as indicated by $\mu_I = W_1 \cdot 2^{k_1} = 1 \cdot 2^1 = 2$.

Let us now consider each LMD d_i in D. Recall that each d_i is of the form $2^{\kappa(x,d_i)} \cdot x \neq t_{d_i}$. Note that d_i constrains only slice $x[0 : p - \kappa(x,d_i) - 1]$. It can be observed that for every combination of values of variables other than x, there is exactly one way of choosing value for slice $x[0 : p - \kappa(x,d_i) - 1]$ such that d_i is violated. This means that there are $2^{\kappa(x,d_i)}$ ways of choosing values for slice_0 through slice_{r+1} such that d_i is violated. Thus for every combination of values of variables other than x, $\sum_{i=1}^{m}(2^{\kappa(x,d_i)})$ is an over-approximation of the number ways of choosing values for slice_0 through slice_{r+1} such that D is violated. Let us denote $\sum_{i=1}^{m}(2^{\kappa(x,d_i)})$ by μ_D. We have already seen that there are at least μ_I ways of adapting an arbitrary solution θ of C to satisfy $C \wedge Z_0 \wedge \ldots \wedge Z_r$. As μ_D is an over-approximation of the number of such adapted solutions that can violate D, there are at least $\mu_I - \mu_D$ ways of adapting θ to satisfy $C \wedge Z_0 \wedge \ldots \wedge Z_r \wedge D$.

Example (Continued): In the example, we have, $d_1 \equiv (x \neq z+7)$ and $\kappa(x,d_1) = 0$. Note that for every value of $z+7$, there is exactly one way of choosing value for slice $x[0:2]$ such that d_1 is violated. $\mu_D = 2^{\kappa(x,d_1)} = 1$, and hence $\mu_I - \mu_D = 1$.

Thus there is at least one way of adapting an arbitrary solution of $(z = 4x + y)$ to satisfy $(z = 4x + y) \wedge (0 \leq 6x + y \leq 4) \wedge (x \neq z + 7)$.

It can be observed that the above reasoning can be extended to the general case $k_1 \geq \ldots \geq k_r$. Let π_i for $0 \leq i < r$ be the number of Z_j's with $k_j < k_i$ for $i < j \leq r$. Using the Inclusion-Exclusion principle, W_{i+1} above then changes to $(\sum_{j=i+1}^{r} \alpha_{i,j}) - (\pi_i - 1) \cdot 2^{k_i - k_{i+1}}$.

Theorem 1. *If* $\eta = \mu_I - \mu_D \geq 1$, *then* $\exists x.\,(C \wedge D \wedge I) \equiv \exists x.\,(C)$

As mentioned earlier, the procedure *QE1_Layer2* applies this technique to problem instances of the form $\exists x.C_2$, obtained after invoking *QE1_Layer1* to find unconstraining LMDs and LMIs. If all the LMIs and LMDs in $\exists x.C_2$ are unconstraining, then $\exists x.C_2$ reduces to $\exists x.\,(2^{k_1} \cdot x = t_1)$, and *QE1_Layer2* returns the equivalent form $2^{p-k_1} \cdot t_1 = 0$.

Example (Continued): *QE1_Layer2* drops the LMI $(6x + y \leq 4)$ and the LMD $(x \neq z + 7)$ as they are unconstraining in $\exists x.\,((z = 4x + y) \wedge (6x + y \leq 4) \wedge (x \neq z + 7))$. The problem instance thus reduces to $\exists x.\,(z = 4x + y)$, which is equivalent to $(4y + 4z = 0)$. Hence the final result is $(4y + 4z = 0)$.

In general, *QE1_Layer2* returns $\exists x.C_3$, where C_3 is a conjunction of possibly fewer LMCs compared to C_2, such that $\exists x.C_3 \equiv \exists x.C_2$. The next section describes techniques to eliminate quantifiers from such problem instances.

***Analysis of Complexity*:** Consider a conjunction of LMCs with a subset of variables in its support to be eliminated. Let n be the number of LMCs in the conjunction, v be the number of variables in its support, and e be the number of variables to be eliminated. Consider the elimination of a variable x inside Layer2. Recall that Layer2 can be applied only when all LMIs involving x are of the form $s \bowtie t$, where $\bowtie \in \{\leq, \geq\}$, s is a linear term with x in its support, and t is a linear term free of x.

Let r be the number of distinct linear terms with x in the support appearing in the LMIs. As observed above, computing η requires $O(r^2)$ arithmetic operations in the worst-case. Note that $r \leq n$. Assuming that each arithmetic operation on p-bit numbers take time $O(Q(p))$ in the worst-case, where $p \leq Q(p) \leq p^3$, elimination of a variable hence has a worst-case time complexity of $O(n^2 \cdot Q(p))$. Observe that eliminating a variable does not increase the number of LMCs in the conjunction. Hence eliminating e variables has a worst-case time complexity of $O(e \cdot n^2 \cdot Q(p))$. Since reading n LMCs as input and writing the result takes $O(n \cdot v \cdot p)$ time, Layer2 has a worst-case time complexity of $O(e \cdot n^2 \cdot Q(p) + n \cdot p \cdot v)$.

3.4 Layer3: Fourier-Motzkin Elimination for LMIs

In this section, we present a Fourier-Motzkin (FM) style QE algorithm for computing $\exists x.C_3$ obtained above. Recall that C_3 obtained above, in general, contains LMDs, LMIs, and a single LME. We propose converting the LMDs and the LME in C_3 to LMIs using the equivalences $(t_1 = t_2) \equiv (t_1 \geq t_2) \wedge (t_1 \leq t_2)$ and $(t_1 \neq t_2)$ $\equiv \neg(t_1 = t_2)$. This, in general, converts C_3 to a Boolean combination of LMIs. However, as we will see in Chapter 4, a QE algorithm for conjunctions of LMCs can be extended to a QE algorithm for Boolean combinations of LMCs. Hence, in the remainder of this section, we will focus on QE from *conjunctions of LMIs*.

There are two fundamental problems when trying to apply FM elimination for reals to a conjunction of LMIs:

1. *Wrap-around behaviour:* Recall that FM elimination normalizes each inequality l w.r.t. the variable x being quantified by expressing l in an equivalent form $x \bowtie t$, where $\bowtie \in \{\leq, \geq\}$ and t is a term free of x. However, due

to wrap-around behaviour, the equivalences (i) $(t_1 \leq t_2) \equiv (t_1 + t_3 \leq t_2 + t_3)$ and (ii) $(t_1 \leq t_2) \equiv (a \cdot t_1 \leq a \cdot t_2)$ used for normalizing inequalities do not hold for LMIs in general. For example, $(2 \leq 3 \pmod 4)$, but $(2+1 > 3+1 \pmod 4)$. Similarly, $(1 \leq 2 \pmod 4)$, but $(1 \cdot 2 > 2 \cdot 2 \pmod 4)$. Hence, normalizing an LMI w.r.t. a variable is much more difficult than normalizing in the case of reals. Moreover, unlike in the case of reals and integers, presence of equalities does not always simplify QE in modular arithmetic. For example, as observed in Section 2.2, $\exists x.\,((2x = 3y+2) \wedge (3x > 4z+3))$ can be simplified to $\exists x.\,((6x = 9y+6) \wedge (6x > 8z+6))$ on integers. However this simplification cannot be done in modular arithmetic in general.

2. *Lack of density:* Even if we could normalize LMIs w.r.t. the variable being quantified, due to the lack of density of integers, FM elimination cannot be directly lifted to normalized LMIs. For example $\exists x.\,((y \leq 4x) \wedge (4x \leq z))$ is equivalent to $(y \leq z)$ in reals, whereas this is not true in modular arithmetic with modulus 2^p, where $p \geq 3$.

This motivates us to (i) define a (weak) normal form for LMIs, and (ii) adapt FM elimination to achieve QE from normalized LMIs. Recall that Omega Test (see Subsection 2.2.2) also defines a normal form for inequalities over integers, and adapts FM elimination over reals for QE from normalized inequalities over integers. However, Omega Test cannot be directly used for QE from LMIs – using Omega Test for QE from LMIs requires converting the LMIs to equivalent constraints in $\mathcal{T}_{\mathcal{Z}}$; the resulting formula is in $\mathcal{T}_{\mathcal{Z}}$, and converting the resulting formula back to modular arithmetic is difficult. Moreover our experiments indicate that, using Omega Test for QE from the $\mathcal{T}_{\mathcal{Z}}$ constraints arising from LMIs incurs considerable performance overhead.

A (weak) normal form for LMIs: We say that an LMI l with x in its support is *normalized w.r.t.* x if it is of the form $a \cdot x \bowtie t$, or of the form $a \cdot x \bowtie b \cdot x$, where $\bowtie \in \{\leq, \geq\}$, and t is a linear term free of x. We will henceforth use $NF1$ to refer to the first normal form ($a \cdot x \bowtie t$) and $NF2$ to refer to the second normal form ($a \cdot x \bowtie b \cdot x$). A Boolean combination of LMCs φ is said to be normalized w.r.t. x if every LMI in φ with x in its support is normalized w.r.t. x.

We will now show that every LMI with x in its support can be equivalently expressed as a Boolean combination of LMCs normalized w.r.t. x. Before going into the details of normalizing LMIs, it would be useful to introduce some notation. We define $\Omega(t_1, t_2)$ as the condition under which $t_1 + t_2$ overflows a p-bit representation, i.e., $t_1 + t_2$ interpreted as an integer exceeds $2^p - 1$. Note that $\Omega(t_1, t_2)$ is equivalent to both $(t_2 \neq 0) \wedge (t_1 \geq -t_2)$ and $(t_1 \neq 0) \wedge (t_2 \geq -t_1)$.

Example: Suppose we wish to normalize the LMI $(x + 2 \leq y)$ modulo 8 w.r.t. x. Adding the additive inverse of 2 modulo 8, i.e, 6 to both sides of the LMI, the left-hand side $x + 2$ changes to x and the right-hand side y changes to $y + 6$. However, note that $(x + 2 \leq y)$ is not equivalent to $(x \leq y + 6)$. If $\Omega(x + 2, 6) \equiv \Omega(y, 6)$, then $(x + 2 \leq y) \equiv (x \leq y + 6)$ holds; otherwise $(x + 2 \leq y) \equiv (x > y + 6)$ holds. Note that $\Omega(x + 2, 6) \equiv \Omega(y, 6)$ can be equivalently expressed as $(x \leq 5) \equiv (y \geq 2)$. Hence, $(x + 2 \leq y)$ can be equivalently expressed in the normalized form $\mathsf{ite}(\varphi, (x \leq y + 6), (x > y + 6))$, where φ denotes $(x \leq 5) \equiv (y \geq 2)$, and $\mathsf{ite}(\alpha, \beta, \gamma)$ is a shorthand for $(\alpha \wedge \beta) \vee (\neg\alpha \wedge \gamma)$.

In this example, the Ω predicate allowed us to perform a case-split and normalize each branch. The following Lemma generalizes this idea.

Lemma 5. *Let $l_1 : (a \cdot x + t_1 \leq b \cdot x + t_2)$ be an LMI, where t_1 and t_2 are linear terms without x in their supports. Then, $l_1 \equiv \mathsf{ite}(\varphi, l_2, \neg l_2)$, where $l_2 \equiv (a \cdot x - b \cdot x \leq$*

$t_2 - t_1$), *and* φ *is a Boolean combination of LMCs normalized w.r.t. x.*

Before we present the proof of Lemma 5, it would be useful to present a proposition.

Proposition 6. *Let* l_1 *be an LMI* $t_1 \leq t_2$*, and let* t_3 *be a linear term. Then* $l_1 \equiv ite(\varphi_1 \wedge (\varphi_2 \oplus \varphi_3), (t_1 + t_3 > t_2 + t_3), (t_1 + t_3 \leq t_2 + t_3))$*, where* $\varphi_1 \equiv (t_3 \neq 0)$*,* $\varphi_2 \equiv (-t_3 \leq t_1)$*,* $\varphi_3 \equiv (-t_3 \leq t_2)$ *and* $\varphi_2 \oplus \varphi_3$ *denotes exclusive-or of* φ_2 *and* φ_3*.*

Proof of Proposition 6. Note that $(t_1 \leq t_2) \equiv \psi_1 \vee \psi_2 \vee \psi_3 \vee \psi_4$, where

- $\psi_1 \equiv (t_1 \leq t_2) \wedge \Omega(t_1, t_3) \wedge \Omega(t_2, t_3)$
- $\psi_2 \equiv (t_1 \leq t_2) \wedge \Omega(t_1, t_3) \wedge \neg\Omega(t_2, t_3)$
- $\psi_3 \equiv (t_1 \leq t_2) \wedge \neg\Omega(t_1, t_3) \wedge \Omega(t_2, t_3)$
- $\psi_4 \equiv (t_1 \leq t_2) \wedge \neg\Omega(t_1, t_3) \wedge \neg\Omega(t_2, t_3)$

It can be seen that,

- $\psi_1 \equiv (t_1 + t_3 \leq t_2 + t_3) \wedge \Omega(t_1, t_3) \wedge \Omega(t_2, t_3)$
- $\psi_2 \equiv$ false, since $\Omega(t_1, t_3) \wedge \neg\Omega(t_2, t_3) \Rightarrow (t_1 > t_2)$. However, we can write ψ_2 as $(t_1 + t_3 > t_2 + t_3) \wedge \Omega(t_1, t_3) \wedge \neg\Omega(t_2, t_3)$ as well, which is equivalent to false, since $\Omega(t_1, t_3) \wedge \neg\Omega(t_2, t_3) \Rightarrow (t_1 + t_3 < t_2 + t_3)$.
- $\psi_3 \equiv (t_1 + t_3 > t_2 + t_3) \wedge \neg\Omega(t_1, t_3) \wedge \Omega(t_2, t_3)$
- $\psi_4 \equiv (t_1 + t_3 \leq t_2 + t_3) \wedge \neg\Omega(t_1, t_3) \wedge \neg\Omega(t_2, t_3)$

Expressing $\psi_1 \vee \psi_2 \vee \psi_3 \vee \psi_4$ in terms of ites, we have,

$$(t_1 \leq t_2) \quad \equiv \quad \mathsf{ite}(\Omega(t_1,t_3) \oplus \Omega(t_2,t_3), (t_1+t_3 > t_2+t_3), (t_1+t_3 \leq t_2+t_3))$$

Expanding the Ω's using the formula $\Omega(\alpha,\beta) \equiv (\beta \neq 0) \wedge (\alpha \geq -\beta)$, where α, β are linear terms, we have,

$$(t_1 \leq t_2) \quad \equiv \quad \mathsf{ite}(\varphi_1 \wedge (\varphi_2 \oplus \varphi_3), (t_1+t_3 > t_2+t_3), (t_1+t_3 \leq t_2+t_3))$$

where, $\varphi_1 \equiv (t_3 \neq 0)$, $\varphi_2 \equiv (-t_3 \leq t_1)$, and $\varphi_3 \equiv (-t_3 \leq t_2)$. □

We can now prove Lemma 5.

Proof of Lemma 5. Consider an LMI $l_1 : a \cdot x + t_1 \leq b \cdot x + t_2$, where t_1 and t_2 are linear terms without x in their supports. Using Proposition 6, with $a \cdot x + t_1$ in place of t_1, $b \cdot x + t_2$ in place of t_2 and $-b \cdot x - t_1$ in place of t_3,

$$l_1 \quad \equiv \quad \mathsf{ite}(\varphi_1 \wedge (\varphi_2 \oplus \varphi_3), (a \cdot x - b \cdot x > t_2 - t_1), (a \cdot x - b \cdot x \leq t_2 - t_1))$$

where, $\varphi_1 \equiv (b \cdot x + t_1 \neq 0)$, $\varphi_2 \equiv (b \cdot x + t_1 \leq a \cdot x + t_1)$, and $\varphi_3 \equiv (b \cdot x + t_1 \leq b \cdot x + t_2)$.

Note that the LMIs $(a \cdot x - b \cdot x > t_2 - t_1)$ and $(a \cdot x - b \cdot x \leq t_2 - t_1)$ are normalized w.r.t. x, whereas φ_2 and φ_3 are not. Hence, let us try to normalize φ_2 and φ_3 w.r.t. x.

Consider $\varphi_2 \equiv (b \cdot x + t_1 \leq a \cdot x + t_1)$. Using Proposition 6, with $b \cdot x + t_1$ in place of t_1, $a \cdot x + t_1$ in place of t_2 and $-t_1$ in place of t_3,

$$\varphi_2 \quad \equiv \quad ite((t_1 \neq 0) \wedge ((t_1 \leq a \cdot x + t_1) \oplus (t_1 \leq b \cdot x + t_1)), (b \cdot x > a \cdot x), (b \cdot x \leq a \cdot x))$$

Using the observations $(\beta \leq \alpha + \beta) \equiv \neg\Omega(\alpha,\beta)$ and $\Omega(\alpha,\beta) \equiv (\beta \neq 0) \wedge (\alpha \geq -\beta)$ for linear terms α and β, and simplifying, $(t_1 \neq 0) \wedge ((t_1 \leq a \cdot x + t_1) \oplus (t_1 \leq b \cdot x +$

$t_1))$ is equivalent to $(t_1 \neq 0) \wedge ((-t_1 \leq a \cdot x) \oplus (-t_1 \leq b \cdot x))$. Hence,

$$\varphi_2 \equiv ite((t_1 \neq 0) \wedge ((-t_1 \leq a \cdot x) \oplus (-t_1 \leq b \cdot x)), (b \cdot x > a \cdot x), (b \cdot x \leq a \cdot x))$$

Similarly, consider $\varphi_3 \equiv (b \cdot x + t_1 \leq b \cdot x + t_2)$. Using Proposition 6, with $b \cdot x + t_1$ in place of t_1, $b \cdot x + t_2$ in place of t_2 and $-b \cdot x$ in place of t_3,

$$\begin{aligned} \varphi_3 &\equiv ite((b \cdot x \neq 0) \wedge ((b \cdot x \leq b \cdot x + t_1) \oplus (b \cdot x \leq b \cdot x + t_2)), (t_1 > t_2), (t_1 \leq t_2)) \\ &\equiv ite((b \cdot x \neq 0) \wedge ((-b \cdot x \leq t_1) \oplus (-b \cdot x \leq t_2)), (t_1 > t_2), (t_1 \leq t_2)) \end{aligned}$$

Putting everything together,

$$\begin{aligned} l_1 &\equiv ite(\varphi_1 \wedge (\varphi_2 \oplus \varphi_3), (a \cdot x - b \cdot x > t_2 - t_1), (a \cdot x - b \cdot x \leq t_2 - t_1)), where \\ \varphi_1 &\equiv (b \cdot x + t_1 \neq 0) \\ \varphi_2 &\equiv ite((t_1 \neq 0) \wedge ((-t_1 \leq a \cdot x) \oplus (-t_1 \leq b \cdot x)), (b \cdot x > a \cdot x), (b \cdot x \leq a \cdot x)) \\ \varphi_3 &\equiv ite((b \cdot x \neq 0) \wedge ((-b \cdot x \leq t_1) \oplus (-b \cdot x \leq t_2)), (t_1 > t_2), (t_1 \leq t_2)) \end{aligned}$$

Hence l_1 can be equivalently expressed as, $\text{ite}(\varphi, l_2, \neg l_2)$, where $l_2 \equiv (a \cdot x - b \cdot x \leq t_2 - t_1)$, and $\varphi \equiv \neg\varphi_1 \vee (\varphi_2 \equiv \varphi_3)$. Note that φ here is a Boolean combination of LMCs normalized w.r.t. x. □

Modified FM for normalized LMIs: We begin by illustrating the primary idea through an example.

Example: Consider the problem of computing $\exists x.C$, where $C \equiv (y \leq 4x) \wedge (4x \leq z)$ with modulus 16. Note that $\exists x.C$ is "the condition under which there exists a multiple of 4 between y and z, where $y \leq z$". Note that if x, y, z were reals, then we would have obtained $(y \leq z)$ for $\exists x.C$. However, as in the case of integers, this would over-approximate $\exists x.C$ in the case of fixed width bit-vectors.

If $(y \leq 12) \wedge (z \geq y+3)$ holds, then the difference between y and z is ≥ 3. In this case, existence of a multiple of 4 between y and z is guaranteed. Thus $(y \leq z) \wedge (y \leq 12) \wedge (z \geq y+3) \Rightarrow \exists x.C$. Note that this case is conceptually similar to *dark shadow* in Omega test.

It can be seen that if $(y > 12)$, then there does not exist any x such that $(y \leq 4x)$. Hence, if $(y > 12)$, then $\exists x.C$ is false. If $(z < y+3)$, then $\exists x.C$ is true iff one of the following conditions holds: (i) $(y \leq z)$ and y is a multiple of 4, i.e., $(y \leq z) \wedge (4y = 0)$, (ii) $(y \leq z)$ and $(y > z \pmod 4)$, i.e., $(y \leq z) \wedge (4y > 4z)$.

Hence $\exists x.C$ is equivalent to $(y \leq z) \wedge \varphi$, where φ is the disjunction of the following three formulas: (i) $(z \geq y+3) \wedge (y \leq 12)$, (ii) $(z < y+3) \wedge (4y = 0)$, (iii) $(z < y+3) \wedge (4y > 4z)$.

The following Lemma generalizes this idea.

Lemma 6. *Let $l_1 : (t_1 \leq a \cdot x)$ and $l_2 : (a \cdot x \leq t_2)$ be LMIs in NF1 w.r.t. x. Let k be $\kappa(x, a \cdot x)$. Then, $\exists x.(l_1 \wedge l_2) \equiv (t_1 \leq t_2) \wedge \varphi$, where φ is the disjunction of the formulas: (i) $(2^{p-k} \cdot t_1 = 0)$, (ii) $(t_2 \geq t_1 + 2^k - 1) \wedge (t_1 \leq 2^p - 2^k)$, and (iii) $(t_2 < t_1 + 2^k - 1) \wedge (2^{p-k} \cdot t_1 > 2^{p-k} \cdot t_2)$.*

Proof of Lemma 6. Note that $\exists x.(l_1 \wedge l_2) \equiv \exists x.(l'_1 \wedge l'_2)$, where $l'_1 \equiv (t_1 \leq 2^k \cdot x)$ and $l'_2 \equiv (2^k \cdot x \leq t_2)$, since the multiples of 2^k and $2^k \cdot e$ are the same modulo 2^p for any odd number $e \in \{1, \ldots, 2^p - 1\}$.

Now $\exists x.(l'_1 \wedge l'_2) \equiv \exists x.\psi_1 \vee \exists x.\psi_2 \vee \exists x.\psi_3 \vee \exists x.\psi_4$, where

- $\psi_1 \equiv l'_1 \wedge l'_2 \wedge (2^{p-k} \cdot t_1 = 0)$
- $\psi_2 \equiv l'_1 \wedge l'_2 \wedge (2^{p-k} \cdot t_1 \neq 0) \wedge (t_2 \geq t_1 + 2^k - 1) \wedge (t_1 \leq 2^p - 2^k)$
- $\psi_3 \equiv l'_1 \wedge l'_2 \wedge (2^{p-k} \cdot t_1 \neq 0) \wedge (t_2 < t_1 + 2^k - 1)$

– $\psi_4 \equiv l_1' \wedge l_2' \wedge (2^{p-k} \cdot t_1 \neq 0) \wedge (t_1 > 2^p - 2^k)$

Consider $\exists x.\psi_1$. This is equivalent to $\exists x.(\psi_1 \wedge (t_1 \leq t_2))$, since $(t_1 \leq t_2)$ is an LMI implied by ψ_1. It can be seen that $\exists x.(\psi_1 \wedge (t_1 \leq t_2))$ is equivalent to $(2^{p-k} \cdot t_1 = 0) \wedge (t_1 \leq t_2)$, since given any solution to $(2^{p-k} \cdot t_1 = 0) \wedge (t_1 \leq t_2)$, we can satisfy $l_1' \wedge l_2'$ by setting $2^k \cdot x = t_1$. Note that setting $2^k \cdot x = t_1$ is always possible, since $2^{p-k} \cdot t_1 = 0 \Rightarrow \exists x.(2^k \cdot x = t_1)$ (see Proposition 3). Hence, $\exists x.\psi_1 \equiv (2^{p-k} \cdot t_1 = 0) \wedge (t_1 \leq t_2)$.

Consider $\exists x.\psi_2$. Note that the difference between t_1 and t_2 here is $\geq 2^k - 1$, which implies that there exists a multiple of 2^k between t_1 and t_2. Hence it can be seen that $(t_1 \leq t_2) \wedge (2^{p-k} \cdot t_1 \neq 0) \wedge (t_2 \geq t_1 + 2^k - 1) \wedge (t_1 \leq 2^p - 2^k) \Rightarrow \exists x.\psi_2$. Implication in the other direction is obvious. Hence, $\exists x.\psi_2 \equiv (t_1 \leq t_2) \wedge (2^{p-k} \cdot t_1 \neq 0) \wedge (t_2 \geq t_1 + 2^k - 1) \wedge (t_1 \leq 2^p - 2^k)$.

Consider $\exists x.\psi_3$. This implies $(2^{p-k} \cdot t_1 > 2^{p-k} \cdot t_2)$. Hence $\exists x.\psi_3 \equiv \exists x.(\psi_3 \wedge (2^{p-k} \cdot t_1 > 2^{p-k} \cdot t_2))$. This is equivalent to $(t_1 \leq t_2) \wedge (2^{p-k} \cdot t_1 \neq 0) \wedge (t_2 < t_1 + 2^k - 1) \wedge (2^{p-k} \cdot t_1 > 2^{p-k} \cdot t_2)$, as the existence of a multiple of 2^k between t_1 and t_2 is implied by $(t_1 \leq t_2) \wedge (2^{p-k} \cdot t_1 \neq 0) \wedge (t_2 < t_1 + 2^k - 1) \wedge (2^{p-k} \cdot t_1 > 2^{p-k} \cdot t_2)$.

Consider $\exists x.\psi_4$. This is equivalent to false, since given $(t_1 > 2^p - 2^k)$, there exists no t_2 such that $l_1' \wedge l_2'$ holds.

Putting everything together, it can be seen that $\exists x.(l_1 \wedge l_2) \equiv (t_1 \leq t_2) \wedge \varphi$, where φ is the disjunction of the formulas: (i) $(2^{p-k} \cdot t_1 = 0)$, (ii) $(t_2 \geq t_1 + 2^k - 1) \wedge (t_1 \leq 2^p - 2^k)$, and (iii) $(t_2 < t_1 + 2^k - 1) \wedge (2^{p-k} \cdot t_1 > 2^{p-k} \cdot t_2)$. □

Suppose we wish to compute $\exists x.I$, where I is a conjunction of LMIs normalized w.r.t. x. Let $I \equiv I_1 \wedge I_2$, where I_1 is the conjunction of LMIs in I that are in *NF1*, and I_2 is the conjunction of LMIs in I that are in *NF2*. In addition, let

$a_1, \ldots, a_n$ be the distinct non-zero coefficients of x in LMIs in I_1, and let $I_{1,i}$ denote the conjunction of LMIs in I_1 in which the coefficient of x is a_i. Finally, let $\Delta(t_1, t_2, k)$ denote the result of computing $\exists x.\,((t_1 \leq a \cdot x) \wedge (a \cdot x \leq t_2))$ using Lemma 6, where k denotes $\kappa(x, a \cdot x)$. It is easy to see that Lemma 6 can be used to compute $\exists x.\, I_{1,i}$, for every $i \in \{1, \ldots n\}$. Similar to FM elimination, we partition the LMIs $l_{i,j} : a_i \cdot x \bowtie t_j$ in $I_{1,i}$ into two sets $\Lambda_{\leq}$ and $\Lambda_{\geq}$, where $\Lambda_{\bowtie} = \{l_{i,j} \mid l_{i,j}$ is of the form $a_i \cdot x \bowtie t_j\}$, for $\bowtie \in \{\leq, \geq\}$. We assume without loss of generality that the trivial LMIs $a_i \cdot x \leq 2^p - 1$ and $a_i \cdot x \geq 0$ are present in $\Lambda_{\leq}$ and $\Lambda_{\geq}$ respectively. We can now compute $\exists x.\, I_{1,i}$ as $\bigwedge_{(a_i \cdot x \leq t_p) \in \Lambda_{\leq},\, (a_i \cdot x \geq t_q) \in \Lambda_{\geq}} \left(\Delta\left(t_q, t_p, \kappa(x, a_i \cdot x)\right)\right)$.

Each conjunction of LMIs such as $I_{1,i}$ above, where all LMIs are in *NF1* w.r.t. x, and have the same coefficient of x are said to be "coefficient-matched" w.r.t. x. Similarly, a Boolean combination of LMCs φ is said to be coefficient-matched w.r.t. x if all LMIs in φ with x in their support are in *NF1* w.r.t. x and have the same coefficient of x. In the special case when $I_2 \equiv \mathsf{true}$ and $n = 1$, i.e., when I is a conjunction of LMIs coefficient-matched w.r.t. x, $\exists x.\, I$ reduces to $\exists x.\, I_{1,1}$.

Unfortunately, converting I to coefficient-matched form w.r.t. a variable is inefficient in general. Hence we propose converting I to coefficient-matched form w.r.t. x only in the following cases, where it can be done without much loss of efficiency: (a) $I_2 \equiv \mathsf{true}$, $n = 2$ and $a_2 = -a_1$, and (b) $I_2 \equiv \mathsf{true}$ and every a_i is of the form $2^{k_i} \cdot e$, where e is an odd number in $\{1, \ldots, 2^p - 1\}$ independent of i.

In case (a) above, I can be equivalently expressed as a Boolean combination of LMCs coefficient-matched w.r.t. x by using the following Proposition.

Proposition 7. *$(-t_1 \leq -t_2)$ is equivalent to $(t_1 = 0) \vee ((t_2 \neq 0) \wedge (t_1 \geq t_2))$.*

Example of Proposition 7: Consider the problem of computing $\exists x.\, I$, where $I \equiv (y \leq 2x) \wedge (6x \leq z)$ with modulus 8. Using Proposition 7, $(6x \leq z)$ is equivalent

to $(2x = 0) \vee ((z \neq 0) \wedge (2x \geq -z))$. Thus $\exists x.I$ can be equivalently expressed as $\exists x.\varphi$, where φ is the disjunction of $(y \leq 2x) \wedge (2x = 0)$ and $(y \leq 2x) \wedge (z \neq 0) \wedge (2x \geq -z)$. Note that φ is coefficient-matched w.r.t. x.

Proof of Proposition 7. $(-t_1 \leq -t_2)$ is equivalent to the disjunction of $(t_1 = 0)$ $\wedge$ $(-t_1 \leq -t_2)$ and $(t_1 \neq 0) \wedge (-t_1 \leq -t_2)$. Note that $(t_1 = 0) \wedge (-t_1 \leq -t_2)$ is equivalent to $(t_1 = 0)$. Moreover, $(t_1 \neq 0) \wedge (-t_1 \leq -t_2)$ is equivalent to $(t_1 \neq 0)$ $\wedge$ $(t_2 \neq 0) \wedge (-t_1 \leq -t_2)$, which is equivalent to $(t_1 \neq 0) \wedge (t_2 \neq 0) \wedge (t_1 \geq t_2)$. Hence $(-t_1 \leq -t_2)$ is equivalent to the disjunction of $(t_1 = 0)$ and $(t_1 \neq 0) \wedge (t_2 \neq 0) \wedge (t_1 \geq t_2)$, which can be simplified to $(t_1 = 0) \vee ((t_2 \neq 0) \wedge (t_1 \geq t_2))$. □

We explain the idea behind case (b) by an example before considering the general case.

Example: Consider the problem of computing $\exists x.I$, where $I \equiv (y \leq 2x) \wedge (x \leq z)$ with modulus 8. It can be shown that $x \leq z$ can be equivalently expressed as the disjunction of (i) $\Omega(x,x) \wedge \Omega(z,z) \wedge (2x \leq 2z)$, (ii) $\neg\Omega(x,x) \wedge \neg\Omega(z,z) \wedge (2x \leq 2z)$, and (iii) $\neg\Omega(x,x) \wedge \Omega(z,z)$. Hence, $\exists x.I$ can be equivalently expressed as $\exists x.\varphi'$, where φ' is the disjunction of (i) $\Omega(x,x) \wedge \Omega(z,z) \wedge (2x \leq 2z) \wedge (y \leq 2x)$, (ii) $\neg\Omega(x,x) \wedge \neg\Omega(z,z) \wedge (2x \leq 2z) \wedge (y \leq 2x)$, and (iii) $\neg\Omega(x,x) \wedge \Omega(z,z) \wedge (y \leq 2x)$. Note that $\Omega(x,x)$ and $\Omega(z,z)$ can be equivalently expressed as $x \geq 4$ and $z \geq 4$ respectively. However, on closer inspection, it can be seen that occurrences of $x \geq 4$ in $\exists x.\varphi'$ arising from $\Omega(x,x)$ are unconstraining, and can therefore be dropped. Thus $\exists x.\varphi'$ can be equivalently expressed as $\exists x.\varphi$, where φ is the disjunction of $(2x \leq 2z) \wedge (y \leq 2x)$ and $(z \geq 4) \wedge (y \leq 2x)$. Note that $\exists x.\varphi$ is equivalent to $\exists x.I$ and is coefficient-matched w.r.t. x.

In general, given $\exists x.I$ such that $I_2 \equiv \mathtt{true}$ and the a_i's have the same e (as defined above), we have the following Lemma.

Lemma 7. *Let I_1 be a conjunction of LMIs in* NF1 *w.r.t. x. Let $a_1, \ldots, a_n$ be the distinct non-zero coefficients of x in LMIs in I_1. Let each a_i, for $1 \leq i \leq n$, be of the form $2^{k_i} \cdot e$, where e is an odd number in $\{1, \ldots, 2^p - 1\}$ independent of i. Then, $\exists x.I_1$ can be equivalently expressed as $\exists x.\varphi$, where φ is a Boolean combination of LMCs coefficient-matched w.r.t. x.*

Proof of Lemma 7. Our proof makes use of the following claims.

Claim 1. *An LMI $a \cdot x \leq t$ in* NF1 *can be equivalently expressed as the disjunction of formulas: (i) $\Omega(a \cdot x, a \cdot x) \wedge \Omega(t,t) \wedge (2a \cdot x \leq 2t)$, (ii) $\neg\Omega(a \cdot x, a \cdot x) \wedge \neg\Omega(t,t) \wedge (2a \cdot x \leq 2t)$, and (iii) $\neg\Omega(a \cdot x, a \cdot x) \wedge \Omega(t,t)$.*

Claim 2. *An LMI $a \cdot x \geq t$ in* NF1 *can be equivalently expressed as the disjunction of formulas: (i) $\Omega(a \cdot x, a \cdot x) \wedge \Omega(t,t) \wedge (2a \cdot x \geq 2t)$, (ii) $\neg\Omega(a \cdot x, a \cdot x) \wedge \neg\Omega(t,t) \wedge (2a \cdot x \geq 2t)$, and (iii) $\Omega(a \cdot x, a \cdot x) \wedge \neg\Omega(t,t)$.*

Proof of Claim 1. Note that $(a \cdot x \leq t) \equiv \psi_1 \vee \psi_2 \vee \psi_3 \vee \psi_4$, where

- $\psi_1 \equiv (a \cdot x \leq t) \wedge \Omega(a \cdot x, a \cdot x) \wedge \Omega(t,t)$
- $\psi_2 \equiv (a \cdot x \leq t) \wedge \Omega(a \cdot x, a \cdot x) \wedge \neg\Omega(t,t)$
- $\psi_3 \equiv (a \cdot x \leq t) \wedge \neg\Omega(a \cdot x, a \cdot x) \wedge \Omega(t,t)$
- $\psi_4 \equiv (a \cdot x \leq t) \wedge \neg\Omega(a \cdot x, a \cdot x) \wedge \neg\Omega(t,t)$

It can be seen that,

- $\psi_1 \equiv \Omega(a \cdot x, a \cdot x) \wedge \Omega(t,t) \wedge (2a \cdot x \leq 2t)$
- $\psi_2 \equiv$ false, since $\Omega(a \cdot x, a \cdot x) \wedge \neg\Omega(t,t) \Rightarrow (a \cdot x > t)$
- $\psi_3 \equiv \neg\Omega(a \cdot x, a \cdot x) \wedge \Omega(t,t)$, since $\neg\Omega(a \cdot x, a \cdot x) \wedge \Omega(t,t) \Rightarrow (a \cdot x < t)$

– $\psi_4 \equiv \neg\Omega(a\cdot x, a\cdot x) \wedge \neg\Omega(t,t) \wedge (2a\cdot x \leq 2t)$

Hence the result. □

Proof of Claim 2. Similar to the proof of Claim 1. □

Without loss of generality, let $a_1 > a_2 > \ldots > a_n$, i.e., $2^{k_1}\cdot e > 2^{k_2}\cdot e > \ldots > 2^{k_n}\cdot e$. This implies that (i) $k_1 > k_2 > \ldots > k_n$, and (ii) $a_1 = 2^{k_1-k_i}\cdot a_i$ for $2 \leq i \leq n$.

Now consider each LMI $a_i \cdot x \bowtie t_j$ in I_1, where $2 \leq i \leq n$ and $\bowtie \in \{\leq, \geq\}$. It can be seen that the above Claims can be used to express $a_i \cdot x \bowtie t_j$ as an equivalent Boolean combination of LMCs, involving (i) the LMI $(2a_i \cdot x \bowtie 2t_j)$, (ii) $\Omega(a_i \cdot x, a_i \cdot x)$, and (iii) $\Omega(t_j, t_j)$. Moreover, the above claims can be used repeatedly to express $a_i \cdot x \bowtie t_j$ as an equivalent Boolean combination of LMCs, involving (i) the LMI $(2^{k_1-k_i}a_i \cdot x \bowtie 2^{k_1-k_i}t_j)$, i.e., $(a_1 \cdot x \bowtie 2^{k_1-k_i}t_j)$, (ii) $\Omega(a_i \cdot x, a_i \cdot x)$, $\Omega(2a_i \cdot x, 2a_i \cdot x)$,..., $\Omega(2^{k_1-k_i-1}a_i \cdot x, 2^{k_1-k_i-1}a_i \cdot x)$, and (iii) $\Omega(t_j, t_j)$, $\Omega(2t_j, 2t_j)$,..., $\Omega(2^{k_1-k_i-1}t_j, 2^{k_1-k_i-1}t_j)$.

It can be seen that $\Omega(a_i\cdot x, a_i\cdot x)$, $\Omega(2a_i\cdot x, 2a_i\cdot x)$,..., $\Omega(2^{k_1-k_i-1}a_i\cdot x, 2^{k_1-k_i-1}a_i\cdot x)$ can be equivalently expressed as $(a_i\cdot x \geq 2^{p-1})$, $(2a_i\cdot x \geq 2^{p-1})$, ..., $(2^{k_1-k_i-1}a_i\cdot x \geq 2^{p-1})$ respectively. Similarly $\Omega(t_j, t_j)$, $\Omega(2t_j, 2t_j)$,..., $\Omega(2^{k_1-k_i-1}t_j, 2^{k_1-k_i-1}t_j)$ can be equivalently expressed as $(t_j \geq 2^{p-1})$, $(2t_j \geq 2^{p-1})$, ..., $(2^{k_1-k_i-1}t_j \geq 2^{p-1})$ respectively. Hence I_1 can be equivalently expressed as a Boolean combination of LMCs φ', involving (i) LMIs of the form $(a_1 \cdot x \bowtie 2^{k_1-k_i}\cdot t_j)$, (ii) LMIs of the form $(a_i \cdot x \geq 2^{p-1})$, $(2a_i \cdot x \geq 2^{p-1})$, ..., $(2^{k_1-k_i-1}a_i \cdot x \geq 2^{p-1})$, and (iii) LMIs of the form $(t_j \geq 2^{p-1})$, $(2t_j \geq 2^{p-1})$, ..., $(2^{k_1-k_i-1}t_j \geq 2^{p-1})$.

We can express φ' equivalently as $\bigvee_{\ell=1}^{r} C_\ell$, where each C_ℓ is a conjunction of LMCs. Hence $\exists x.\varphi'$ is equivalent to $\bigvee_{\ell=1}^{r} (\exists x. C_\ell)$. Observe that each C_ℓ involves three kinds of LMIs: (i) LMIs of the form $(a_1 \cdot x \bowtie 2^{k_1-k_i}\cdot t_j)$, (ii) LMIs of the form

$(a_i \cdot x \geq 2^{p-1})$, $(2a_i \cdot x \geq 2^{p-1})$, ..., $(2^{k_1-k_i-1}a_i \cdot x \geq 2^{p-1})$ and/or their negations, and (iii) LMIs of the form $(t_j \geq 2^{p-1})$, $(2t_j \geq 2^{p-1})$, ..., $(2^{k_1-k_i-1}t_j \geq 2^{p-1})$ and/or their negations. Let $C_{\ell,1}$ be the conjunction of the first kind of LMIs in C_ℓ. Similarly, let $C_{\ell,2}$ and $C_{\ell,3}$ respectively be the conjunctions of the second and the third kinds of LMIs in C_ℓ. Hence we have $C_\ell \equiv C_{\ell,1} \wedge C_{\ell,2} \wedge C_{\ell,3}$.

Therefore $\exists x.C_\ell \equiv (\exists x.(C_{\ell,1} \wedge C_{\ell,2})) \wedge C_{\ell,3}$, since $C_{\ell,3}$ is free of x. Moreover, by applying Theorem 1 on $\exists x.(C_{\ell,1} \wedge C_{\ell,2})$, it can be proved that $C_{\ell,2}$ is unconstraining in $\exists x.(C_{\ell,1} \wedge C_{\ell,2})$. Hence $\exists x.C_\ell$ can be equivalently expressed as $\exists x.(C_{\ell,1}) \wedge C_{\ell,3}$. Note that the coefficient of x in $C_{\ell,1}$ is a_1. This implies that $\bigvee_{\ell=1}^{r} C_\ell$ can be equivalently expressed as a Boolean combination of LMCs coefficient-matched w.r.t. x, with coefficient of x as a_1. □

Note that normalizing a given conjunction of LMIs w.r.t. a variable and then converting it to coefficient-matched form transforms it to a Boolean combination of LMCs in general. We make use of one of the techniques in Chapter 4 for eliminating quantifiers from such Boolean combinations of LMCs.

In cases other than those covered in cases (a) and (b) above, we propose computing $\exists x.I$ using *model enumeration*, i.e., by expressing $\exists x.I$ in the equivalent form $I|_{x\leftarrow 0} \vee \ldots \vee I|_{x\leftarrow 2^p-1}$ where $I|_{x\leftarrow i}$ denotes I with x replaced by the constant i.

The procedure that computes $\exists x.C_3$ (where C_3 is obtained from *QE1_Layer2*) using techniques mentioned in this section is called *QE1_Layer3* (see Algorithm 1). Initially, the LMDs and the single LME in the conjunction are converted to LMIs using the equivalences $(t_1 = t_2) \equiv (t_1 \geq t_2) \wedge (t_1 \leq t_2)$ and $(t_1 \neq t_2) \equiv \neg(t_1 = t_2)$. This in general yields a Boolean combination of LMCs φ_1. If φ_1 is a conjunction of LMIs coefficient-matched w.r.t. x, then $\exists x.\varphi_1$ is computed using the modified

FM elimination in Lemma 6. Otherwise, $\exists x.\varphi_1$ is computed either by converting φ_1 to coefficient-matched form w.r.t. x, followed by QE from the resulting Boolean combination of LMCs, or by model enumeration.

Algorithm 1: *QE1_Layer3*

Input: Conjunction of LMCs C, Variable to eliminate x

Output: Boolean combination of LMCs ψ equivalent to $\exists x.C$

```
1  φ1 := convertToLMIs(C);          // convert LMEs and LMDs to LMIs
2  if φ1 is a coefficient-matched conjunction w.r.t. x then
3      ψ := modifiedFM(φ1, x);
       // Apply modified FM based on Lemma 6
4  else
5      if model enumeration is selected to compute ∃x.φ1 then
6          ψ := modelEnumerate(φ1, x);      // Apply model enumeration
7      else
8          φ2 := coefficientMatch(φ1, x);
           // φ1 in general is a Boolean combination
9          ψ := QEFromBooleanCombination(φ2, x);
           // Eliminate x from Boolean combination φ2;
           // (see Chapter 4 for details)
10 return ψ;
```

***Analysis of Complexity*:** Consider a conjunction of LMCs with a subset of variables in its support to be eliminated. Let n be the number of LMCs in the conjunction, v be the number of variables in its support, and e be the number of variables

to be eliminated. Note that Layer3 resorts to model enumeration in the worst case. Consider the elimination of the first quantified variable, say, x_1 by model enumeration.

Elimination of x_1 by model enumeration involves creating 2^p copies of the conjunction, and then replacing x_1 by a constant in each copy. Replacing x_1 by constant and then simplifying takes $O(n)$ arithmetic operations in the worst-case for each copy. Assuming that each arithmetic operation on p-bit numbers take time $O(Q(p))$ in the worst-case, where $p \leq Q(p) \leq p^3$, elimination of x_1 from each copy hence has a worst-case time complexity of $O(n \cdot Q(p))$. Since there are 2^p such copies, elimination of x_1 has a worst-case time complexity of $O(n \cdot Q(p) \cdot 2^p)$.

Elimination of x_1 generates a formula with 2^p disjuncts, where each disjunct can have n LMCs. In a similar manner as above, it can be seen that elimination of the second quantified variable, say, x_2 has a worst-case time complexity of $O(n \cdot Q(p) \cdot 2^{2 \cdot p})$. Proceeding like this, it can be seen that elimination of e quantified variables has a worst-case time complexity of $O\left(n \cdot Q(p) \cdot \left(2^p + 2^{2 \cdot p} + \ldots 2^{e \cdot p}\right)\right)$, which reduces to $O(n \cdot Q(p) \cdot 2^{(e+1) \cdot p})$.

After elimination of e variables, we have a formula with $2^{e \cdot p}$ disjuncts, where each disjunct can have n LMCs. Writing each disjunct involving n LMCs takes $O(n \cdot v \cdot p)$ time. Hence writing the result takes $O(n \cdot v \cdot p \cdot 2^{e \cdot p})$ time. Therefore Layer3 has a worst-case time complexity of $O(n \cdot Q(p) \cdot 2^{(e+1) \cdot p} + n \cdot v \cdot p \cdot 2^{e \cdot p})$.

3.5 *Project*: Combining Layers

Recall that *QE1_Layer1*, *QE1_Layer2*, and *QE1_Layer3* try to eliminate a single quantifier from a conjunction of LMCs. These procedures can be extended to

eliminate multiple quantifiers by invoking them iteratively. Thus we have procedures *Layer1*, *Layer2*, and *Layer3* - extensions of *QE1_Layer1*, *QE1_Layer2*, and *QE1_Layer3* respectively, to eliminate multiple quantifiers.

Algorithm 2: *Project*

Input: Conjunction of LMCs A, Set of variables to eliminate X

Output: Boolean combination of LMCs ψ equivalent to $\exists X.A$

```
1  φ1 := Layer1(A, X);          // for each x ∈ X, Apply QE1_Layer1
2  if φ1 has no quantifiers then
3      ψ := φ1;
4  else
       // Let φ1 ≡ A1 ∧ ∃Y.B
5      φ2 := Layer2(B, Y);      // for each x ∈ Y, Apply QE1_Layer2
6      if φ2 has no quantifiers then
7          ψ := A1 ∧ φ2;
8      else
           // Let φ2 ≡ A2 ∧ ∃Z.C
9          φ3 := Layer3(C, Z); // for each x ∈ Z, Apply QE1_Layer3
10         ψ := A1 ∧ A2 ∧ φ3;
11 return ψ;
```

We now present the overall QE algorithm *Project* (see Algorithm 2) for computing $\exists X.A$, where A is a conjunction of LMCs over a set of variables V such that $X \subseteq V$. Initially *Project* tries to compute $\exists X.A$ using *Layer1*. This reduces $\exists X.A$ to an equivalent conjunction of LMCs φ_1. If all variables in X are eliminated

by *Layer1*, then φ_1 is free of quantifiers. In this case, $\exists X.A$ is equivalent to φ_1, and *Project* returns φ_1. Otherwise, φ_1 is equivalent to the conjunction of A_1 and $\exists Y.B$, where A_1, B are conjunctions of LMCs, $Y \subseteq X$, and $X \setminus Y$ is the subset of variables in X that are eliminated by *Layer1*. *Project* then tries to compute $\exists Y.B$ using *Layer2*.

Layer2 reduces $\exists Y.B$ to an equivalent conjunction of LMCs φ_2. If all variables in Y are eliminated by *Layer2*, then φ_2 is free of quantifiers. In this case $\exists X.A$ is equivalent to $A_1 \wedge \varphi_2$, and *Project* returns $A_1 \wedge \varphi_2$. Otherwise, φ_2 is equivalent to the conjunction of A_2 and $\exists Z.C$, where A_2, C are conjunctions of LMCs, $Z \subseteq Y$, and $Y \setminus Z$ is the subset of variables in Y that are eliminated by *Layer2*. *Project* calls *Layer3* to compute $\exists Z.C$. *Layer3* computes φ_3, a Boolean combination of LMCs equivalent to $\exists Z.C$, and *Project* returns $A_1 \wedge A_2 \wedge \varphi_3$.

Let x be the variable being eliminated. Line-8 of *QE1_Layer3* generates a Boolean combination of LMCs φ_2 coefficient-matched w.r.t. x. Line-9 of *QE1_Layer3* then calls *QEFromBooleanCombination* in order to eliminate x from φ_2. This eventually gets reduced to eliminating x from a bunch of conjunctions of LMCs. Eliminating x from each such conjunction of LMCs results in a new recursive *Project* call. Because of this feedback, the control flow inside *Project* is not linear.

Note that each new recursive *Project* call may in turn call *QE1_Layer3*. However it can be observed that this mutual recursion between *QE1_Layer3* and *Project* does not result in infinite recursion. To see this, note that in each of the recursive *Project* calls, all LMIs involving x are coefficient-matched w.r.t. x. Hence x will be certainly eliminated by *Layer1*, *Layer2*, or *modifiedFM* inside these recursive *Project* calls. This guarantees that the recursion terminates.

3.6 Experimental Results

We performed experiments to (i) evaluate the performance and effectiveness of the layers in *Project* and (ii) compare the performance of *Project* with alternative QE techniques. All the experiments were performed on a 1.83 GHz Intel(R) Core 2 Duo machine with 2GB memory running Linux, with a timeout of 1800 seconds. We used the same variable ordering in all experiments using BDDs [14]. We performed depth-first traversal of the Boolean formulas from which the BDDs were created. The variables were ordered in the order they were encountered in the depth-first traversal. In *Project*, inside the layers, when there were multiple variables to eliminate, we used a simple lexicographic variable elimination order. Moreover, inside *Layer3*, the variables with constraints in coefficient-matched form were eliminated before the variables which required transformation to Boolean combination. In experiments involving Omega Test, we used Pugh et al.'s implementation of Omega Test from [68].

Benchmarks: We used a benchmark suite consisting of 198 *lindd* benchmarks [28] and 39 *vhdl* benchmarks. Each of these benchmarks is a Boolean combination of LMCs with a subset of the variables in their support existentially quantified.

The *lindd* benchmarks reported in [28] are Boolean combinations of octagonal constraints over integers, i.e., constraints of the form $a \cdot x + b \cdot y \leq k$ where x, y are integer variables, k is an integer constant, and $a, b \in \{-1, 1\}$. We converted these benchmarks to Boolean combinations of LMCs by assuming the size of integer as 16 bits. Although these benchmarks had no LMEs explicitly, they contained LMEs encoded as conjunctions of the form $(x - y \leq k) \wedge \neg(x - y \leq k - 1)$. We converted each such conjunction to an LME $x - y = k$ as a pre-processing step.

The total number of variables, the number of variables to be eliminated, and the number of bits to be eliminated in the *lindd* benchmarks ranged from 30 to 259, 23 to 207, and 368 to 3312 respectively.

The *vhdl* benchmarks were obtained in the following manner. We took a set of word-level VHDL designs. Some of these are publicly available designs obtained from [69], and the remaining are proprietary. We derived the symbolic transition relations of these VHDL designs. The *vhdl* benchmarks were obtained by quantifying out all the internal variables (i.e. neither input nor output of the top-level module) from these symbolic transition relations. Effectively this gives abstract transition relations of the designs. The coefficients of the variables in these benchmarks were largely odd. These benchmarks contained a significant number of LMEs (arising from assignment statements in the VHDL programs). The total number of variables, the number of variables to be eliminated, and the number of bits to be eliminated in the *vhdl* benchmarks ranged from 8 to 50, 2 to 21, and 10 to 672 respectively.

Evaluation of *Project*: We performed QE from the benchmarks using the algorithms in Chapter 4, and analyzed the *Project* calls that were generated during this process. Recall that *Layer3* involves transforming a conjunction of LMCs to a Boolean combination of LMCs and QE from this Boolean combination. This results in new (recursive) *Project* calls. Hence two kinds of *Project* calls were generated while performing QE from the benchmarks: (i) the initial/original *Project* calls, and the (ii) aforementioned recursive *Project* calls. In the subsequent discussion, whenever we mention "*Project* calls", it refers to the initial/original *Project* calls, unless stated otherwise.

The total number of *Project* calls generated from the *lindd* and *vhdl* bench-

Table 3.1: Details of *Project* calls (figures are per *Project* call)

Type	Vars	Qnt	LMIs	LMEs	LMDs	Contr			Time			
						L1	L2	L3	L1	L2	L3	*Pr*
lindd	39.9	38.1	(88, 0, 18.9)	(60, 0, 10.1)	(35, 0, 8.1)	51	44	5	3	5	13149	674
vhdl	8.6	7.2	(4, 0, 0.3)	(16, 0, 5.8)	(31, 0, 2.0)	95	4.5	0.5	2	6	161	3

Vars : Average number of variables, **Qnt** : Average number of quantifiers, **LMIs** : (Maximum, minimum, average) number of LMIs, **LMEs** : (Maximum, minimum, average) number of LMEs, **LMDs** : (Maximum, minimum, average) number of LMDs, **Contr** : Average contribution of a layer, **L1** : *Layer1*, **L2** : *Layer2*, **L3** : *Layer3*, **Pr** : *Project*, **Time** : Average time spent per quantifier eliminated in milliseconds

marks were $52,836$ and $8,027$ respectively. Statistics of these *Project* calls are shown in Table 3.1. The contribution of a layer is measured as the ratio of the number of quantifiers eliminated by the layer to the number of quantifiers to be eliminated in the *Project* call multiplied by 100. The time spent per quantifier eliminated for a layer is measured as the ratio of the time spent inside the layer to the number of quantifiers eliminated by the layer. The contributions of the layers and the times taken by the layers per quantifier eliminated for individual *Project* calls from *lindd* benchmarks are shown in Fig. 3.2, Fig. 3.3 and Fig. 3.6, and those for individual *Project* calls from *vhdl* benchmarks are shown in Fig. 3.4, Fig. 3.5 and Fig. 3.7. The *Project* calls here are sorted in increasing order of contribution from *Layer1*.

Layer1 and *Layer2* were cheap and eliminated a large fraction of quantifiers in both *lindd* and *vhdl* benchmarks. This underlines the importance of our layered framework. The relatively large contribution of *Layer1* in the *Project* calls from *vhdl* benchmarks was due to significant number of LMEs in these problem instances. *Layer3* was found to be the most expensive layer. Most of the time spent in *Layer3* was consumed in the recursive *Project* calls. No *Layer3* call in our experiments required model enumeration. The large gap in the time per quantifier

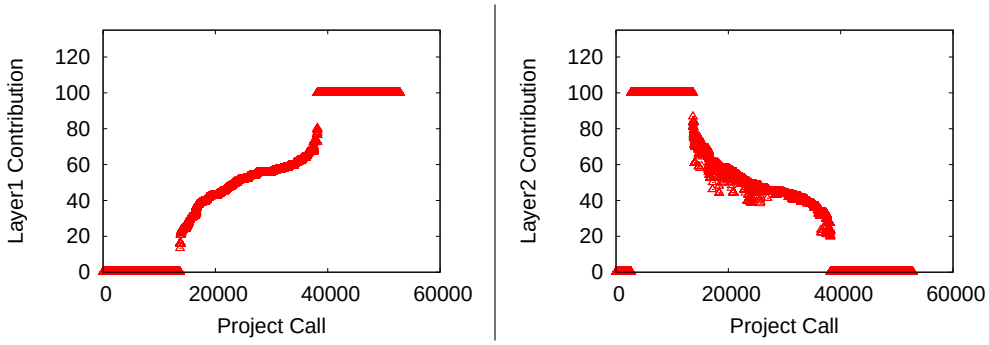

Figure 3.2: Contribution of (a) *Layer1* and (b) *Layer2* for *lindd* benchmarks

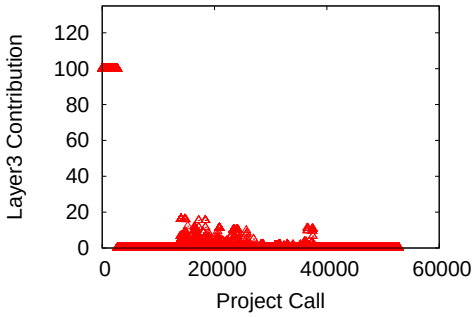

Figure 3.3: Contribution of *Layer3* for *lindd* benchmarks

in *Layer2* and that in *Layer3* for both sets of benchmarks points to the need for developing additional cheap layers between *Layer2* and *Layer3* as part of future work.

Comparison of *Project* with alternative QE techniques: We compared the performance of *Project* with QE based on linear integer arithmetic using Omega Test, and also with QE based on bit-blasting. We implemented the following algorithms for this purpose: (i) *Layer1_Blast*: this procedure first quantifies out the variables using *Layer1* (recall that *Layer1* is a simple extension of the work in [79]), and then uses bit-blasting and BDD based bit-level QE [14] for the remaining variables. (ii) *Layer1_OT*, *Layer2_OT*: *Layer1_OT* first quantifies out the

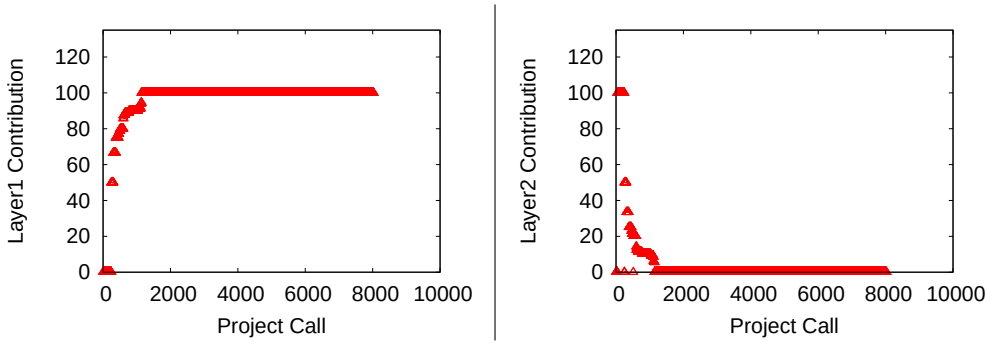

Figure 3.4: Contribution of (a) *Layer1* and (b) *Layer2* for *vhdl* benchmarks

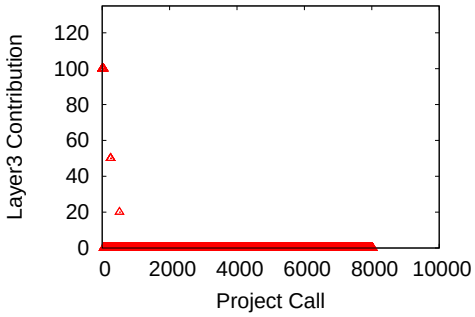

Figure 3.5: Contribution of *Layer3* for *vhdl* benchmarks

variables using *Layer1*, and then uses conversion to linear integer arithmetic and Omega Test for the remaining variables. *Layer2_OT* first quantifies out the variables using *Layer1* followed by *Layer2*, and then uses conversion to linear integer arithmetic and Omega Test for the remaining variables. *Layer2_OT* helps us to compare the performance of *Layer3* with that of Omega Test.

We collected 100 instances of QE problem for conjunctions of LMCs arising from the algorithm *QE_SMT* (see Section 4.2) when QE is performed on the benchmarks. We performed QE from these conjunction-level problem instances using *Project*, *Layer1_Blast*, *Layer1_OT*, and *Layer2_OT*. Fig. 3.8(a) and 3.8(b) compare the QE times taken by *Project* against those taken by *Layer1_Blast* and

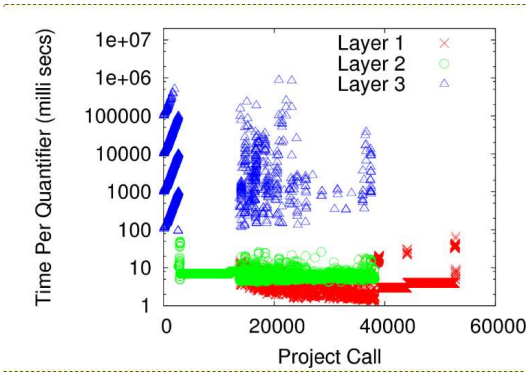

Figure 3.6: Cost of layers for *lindd* benchmarks

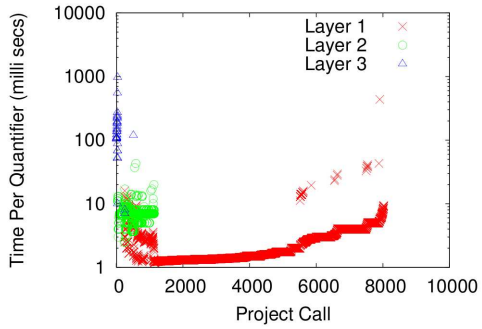

Figure 3.7: Cost of layers for *vhdl* benchmarks

Layer1_OT for each of these conjunction-level problem instances.

Project could successfully eliminate quantifiers in all of the 100 instances. *Layer1_Blast* was unsuccessful in in 68 cases and *Layer1_OT* were unsuccessful in 65 cases. These cases are indicated by the topmost points in Fig. 3.8(a) and 3.8(b) respectively. In most cases where *Layer1_Blast* and *Layer1_OT* were successful, the times taken by all the three algorithms were comparable. However there were a few cases where *Layer1_Blast* and *Layer1_OT* performed better than *Project*.

We found that these cases involved *Layer3*, and most of the time consumed by *Project* was spent inside *Layer3*.

We compared the times consumed by *Layer3* in *Project* with those consumed by Omega Test in *Layer2_OT* (see Fig. 3.9). There were 51 problem instances which required *Layer3*. Omega Test timed out in 37 of them. In 13 of the remaining 14 cases, Omega Test performed better than *Layer3*. Our analysis revealed that these cases were simpler in terms of number of LMCs and number of variables to be eliminated. However *Layer3* incurred several recursive *Project* calls in these cases.

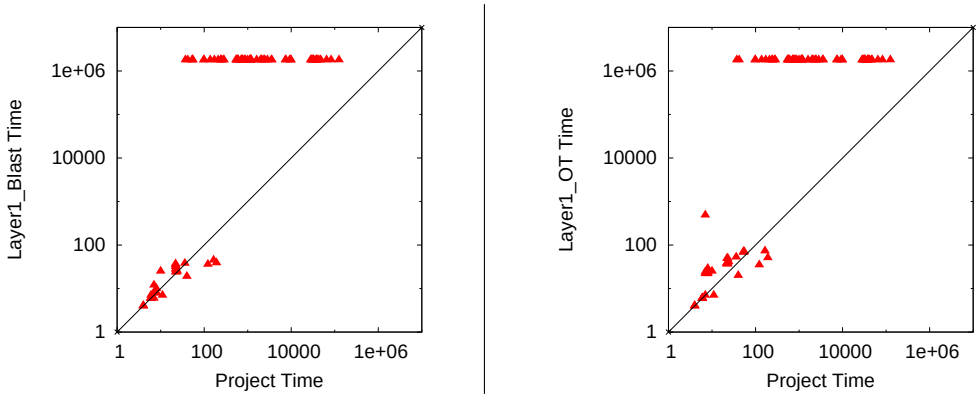

Figure 3.8: Plots comparing (a) *Project* and *Layer1_Blast* and (b) *Project* and *Layer1_OT* (All times are in milliseconds)

Comparison of *Layer2* with alternative techniques: Recall that given $\exists x.(C \wedge D \wedge I)$, where C is a conjunction of LMCs, D is a conjunction of LMDs and I is a conjunction of LMIs, *Layer2* checks if $\exists x.(C) \equiv \exists x.(C \wedge D \wedge I)$ holds. *Layer2* performs this check by computing an efficiently computable under-approximation of the number of ways in which an arbitrary solution of C can be engineered to satisfy $C \wedge D \wedge I$. We compared the performance of *Layer2* with a BDD based alternative

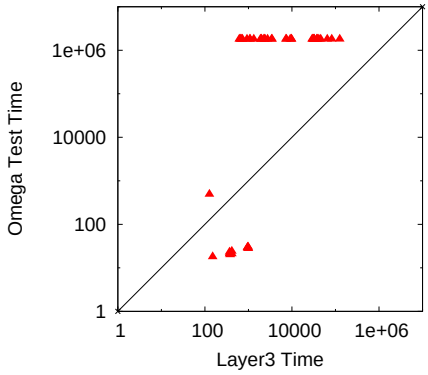

Figure 3.9: Plot comparing *Layer3* and Omega Test (All times are in milliseconds)

technique to perform this check. We implemented a procedure *BddBasedLayer2* for this purpose. *BddBasedLayer2* computes BDDs for $\exists x.(C)$ and $\exists x.(C \wedge D \wedge I)$, and then checks if these BDDs are the same. $\exists x.(C) \equiv \exists x.(C \wedge D \wedge I)$ holds iff the BDDs for $\exists x.(C)$ and $\exists x.(C \wedge D \wedge I)$ are the same. We then implemented procedure *ProjectWithBddBasedLayer2* which is a variant of *Project* that uses *BddBasedLayer2* in place of *Layer2*.

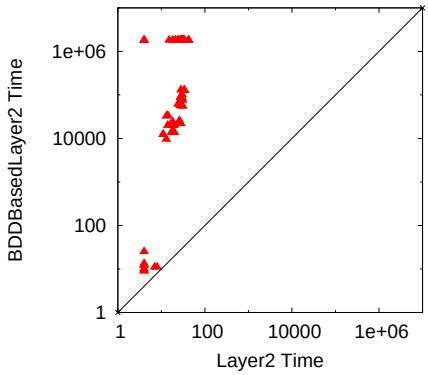

Figure 3.10: Plot comparing *Layer2* and *BddBasedLayer2* (All times are in milliseconds)

We performed QE from the 100 conjunction-level problem instances using

ProjectWithBddBasedLayer2. For each problem instance, we then compared the time consumed by *Layer2* in *Project* with that consumed by *BddBasedLayer2* in *ProjectWithBddBasedLayer2* (see Fig. 3.10). *Layer2* outperformed the BDD based alternative technique in all the 100 problem instances.

3.7 Conclusions

The need for efficient techniques for bit-precise QE cannot be overemphasized. In this chapter, we presented such a bit-precise and practically efficient QE algorithm for conjunctions of LMCs. Our experiments demonstrated that our modular arithmetic based algorithm for QE outperforms linear integer arithmetic and bit-blasting based QE techniques. Moreover, our algorithm keeps the quantifier eliminated formula in modular arithmetic, which allows further modular arithmetic level reasoning on the quantifier eliminated formula. It is interesting to see how we can extend this algorithm to eliminate quantifiers from arbitrary Boolean combinations of LMCs. Next chapter presents results of our investigations in this direction.

Chapter 4

Extending Quantifier Elimination to Boolean Combinations

In Chapter 3, we presented a QE algorithm *Project* for conjunctions of LMCs which is bit-precise and efficient in practice. The motivation behind the development of this algorithm was its applications in formal verification and analysis of word-level RTL designs and embedded programs. However, the symbolic transition relations of word-level RTL designs and embedded programs involve arbitrary Boolean combinations of LMCs, not necessarily conjunctions of LMCs. Hence, the QE problem instances that arise in formal verification and analysis of such designs and programs involve QE from arbitrary Boolean combinations of LMCs. Thus extending *Project* to eliminate quantifiers from arbitrary Boolean combinations of LMCs is an important problem. We address this problem in this chapter.

As a motivating example, consider the synchronous circuit shown in Fig. 4.1, with the relevant part of its functionality described in VHDL in Fig. 4.2. The

circuit comprises a controller and three 8-bit registers, A, B, and X. The controller switches between three states, 0, 1, and 2. In state 0, the values of A and B are read from inputs *InA* and *InB* respectively, and are stored in corresponding registers. In addition, the value of X is initialized to 0, and the control moves to state 1. State 1 implements the iterative algorithm: if $X + A \leq B$, the value of X is incremented, that of A is doubled, and the circuit continues to iterate in state 1. If, however, $X + A > B$, the circuit checks if the value of X equals $B + 1$. If so, the control moves to state 0 via state 2. Otherwise, the control moves directly to state 0 from state 1.

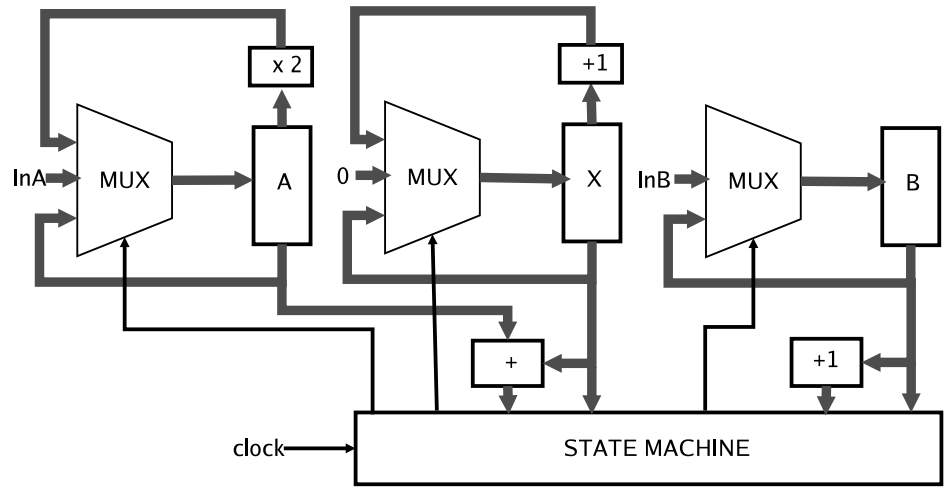

Figure 4.1: An example circuit

The symbolic transition relation, R, for this circuit can be obtained by conjoining the following equality relations, where primed variables refer to values of the

```
 ...
if (clock'event and clock = '1') then
 case state is
  when "00" => A <= InA;
    B <= InB; X <= x"00"; state <= "01";
  when "01" => if (X + A <= B) then
   X <= X+'1'; A <= x"02"*A;
   elsif (X = B+'1') then state <= "10";
   else state <= "00"; end if;
  when others => state <= "00";
 end case;
end if;
   ....
```

Figure 4.2: VHDL program for example circuit

corresponding unprimed variables after the next rising edge of the clock.

$$
\begin{aligned}
\mathsf{state}' &= \mathsf{ite}(\mathsf{state}=0,1,\mathsf{ite}(\mathsf{state}=1,\mathsf{ite}(\mathsf{X}+\mathsf{A}\leq\mathsf{B},1,\mathsf{ite}(\mathsf{X}=\mathsf{B}+1,2,0)),0)) \\
\mathsf{A}' &= \mathsf{ite}(\mathsf{state}=0,\mathsf{InA},\mathsf{ite}(\mathsf{state}=1,\mathsf{ite}(\mathsf{X}+\mathsf{A}\leq\mathsf{B},2\cdot\mathsf{A},\mathsf{A}),\mathsf{A})) \\
\mathsf{B}' &= \mathsf{ite}(\mathsf{state}=0,\mathsf{InB},\mathsf{B}) \\
\mathsf{X}' &= \mathsf{ite}(\mathsf{state}=0,0\mathsf{x}00,\mathsf{ite}(\mathsf{state}=1,\mathsf{ite}(\mathsf{X}+\mathsf{A}\leq\mathsf{B},\mathsf{X}+1,\mathsf{X}),\mathsf{X}))
\end{aligned}
$$

In the above equalities, $\mathsf{A},\mathsf{A}',\mathsf{B},\mathsf{B}',\mathsf{InA},\mathsf{InB},\mathsf{X}$, and X' refer to bit-vectors of width 8, whereas state and state' refer to bit-vectors of width 2. Furthermore, all operations and comparisons involving $\mathsf{A},\mathsf{A}',\mathsf{B},\mathsf{B}',\mathsf{InA},\mathsf{InB},\mathsf{X}$, and X' are unsigned operations modulo 2^8, and those involving state and state' are unsigned opera-

tions modulo 2^2. Since $a = ite(b,c,d)$ represents $(b \wedge (a = c)) \vee (\neg b \wedge (a = d))$, the transition relation R above is a Boolean combination of LMCs.

The above circuit computes the smallest 8-bit non-negative integer X such that $2^{\mathsf{X}} \cdot \mathsf{lnA} + \mathsf{X} > \mathsf{lnB}$, where all the operations are modulo 2^8. If the smallest value of X thus computed is $\mathsf{lnB} + 1$, the control enters state 2; otherwise it returns to state 0. For example, suppose $\mathsf{lnA} = 1$ and $\mathsf{lnB} = 150$. Inside state 1, the value of A overflows to zero after 8 iterations and remains as zero thereafter. The value of X is incremented in each iteration until it becomes 151. Now that $\mathsf{X} + \mathsf{A} \leq \mathsf{B}$ is false and $\mathsf{X} = \mathsf{B} + 1$ is true, the control moves to state 2. Observe that 151 is the smallest 8-bit non-negative integer X such that $2^{\mathsf{X}} \cdot 1 + \mathsf{X} > 150$ modulo 2^8.

This circuit has the property that if it starts in state 0, then the value of A is always less than $255 \cdot \mathsf{X}$ when it visits state 2. The value of A may exceed $255 \cdot \mathsf{X}$ and even overflow during the modulo 2^8 multiplications in state 1. However, when it reaches state 2, A is less than $255 \cdot \mathsf{X}$. To see why this is true, observe that in state 2, both $\mathsf{X} + \mathsf{A} > \mathsf{B}$ and $\mathsf{X} = \mathsf{B} + 1$ are true; hence $\mathsf{X} + \mathsf{A} > \mathsf{X} + 255$ is true, where 255 is the additive inverse of 1 in modulo 2^8. Note that since $\mathsf{A} \leq 255$, $\mathsf{X} + \mathsf{A} > \mathsf{X} + 255$ implies $\mathsf{X} \neq 0$. Moreover, since $\mathsf{A} \leq 255$, if the operation $\mathsf{X} + \mathsf{A}$ overflows, then $\mathsf{X} + \mathsf{A} \leq \mathsf{X} + 255$ holds for $\mathsf{X} \neq 0$. But we have $\mathsf{X} + \mathsf{A} > \mathsf{X} + 255$. Hence the operation $\mathsf{X} + \mathsf{A}$ should not overflow. This implies that A is less than the additive inverse of X modulo 2^8. Since $255 \cdot \mathsf{X}$ is the additive inverse of X modulo 2^8, we have $\mathsf{A} < 255 \cdot \mathsf{X}$.

Suppose we wish to verify this property for the first N time steps of operation of the circuit using bounded model checking. This involves unrolling the transition relation N times, conjoining the unrolled relation with the negation of the property, and feeding the resulting formula to an SMT solver [10]. Observe that R

contains primed and unprimed versions of all variables in the circuit. Hence, unrolling R a large number of times can give a formula with a very large number of variables. While the number of variables in an SMT formula is not the sole determinant of performance of SMT solving, formulas with large numbers of variables typically lead to performance bottlenecks in SMT solving.

A common approach to circumventing this problem is to use an abstract transition relation R' that relates values of only a chosen subset of variables relevant to the property being checked, while abstracting the relation between the other variables. In general, the set of states reached using R' overapproximates the exact set of reachable states. Therefore, if N-step bounded model checking using R' fails to give a counterexample, then the property holds in N steps of operation of the circuit.

In our example, an abstract transition relation R' can be obtained by computing $\exists B \exists B' \exists InB.\ R$. An equivalent quantifier-free version of R' is given below.

$$\begin{aligned}
&((\mathsf{state} = 0) \wedge (\mathsf{state}' = 1) \wedge (\mathsf{A}' = \mathsf{InA}) \wedge (\mathsf{X}' = 0\text{x}00)) \quad \vee \\
&((\mathsf{state} = 1) \wedge (\mathsf{state}' = 1) \wedge (\mathsf{A}' = 2 \cdot \mathsf{A}) \wedge (\mathsf{X}' = \mathsf{X} + 1)) \quad \vee \\
&((\mathsf{state} = 1) \wedge (\mathsf{state}' = 2) \wedge (\mathsf{A}' = \mathsf{A}) \wedge (\mathsf{X}' = \mathsf{X}) \wedge (\mathsf{X} + \mathsf{A} > \mathsf{X} + 255)) \quad \vee \\
&((\mathsf{state} = 1) \wedge (\mathsf{state}' = 0) \wedge (\mathsf{A}' = \mathsf{A}) \wedge (\mathsf{X}' = \mathsf{X}) \wedge \varphi) \quad \vee \\
&((\mathsf{state} \neq 0) \wedge (\mathsf{state} \neq 1) \wedge (\mathsf{state}' = 0) \wedge (\mathsf{A}' = \mathsf{A}) \wedge (\mathsf{X}' = \mathsf{X}))
\end{aligned}$$

where φ is the disjunction of the formulas $(\mathsf{X} + \mathsf{A} \neq 0) \wedge (\mathsf{X} \neq 1)$ and $(\mathsf{X} + \mathsf{A} \neq 0) \wedge (\mathsf{X} \neq 0) \wedge (\mathsf{X} \leq \mathsf{X} + \mathsf{A} + 255)$.

It can indeed be verified that bounded model checking using R' (instead of R) suffices to show that if the circuit starts in state 0, the value of A is always less than $255 \cdot X \pmod{256}$ when it visits state 2. Since R' does not contain B, B' or InB, the number of variables in N unrollings of R' is less than that in N

unrollings of R. This often leads to better performance of SMT solving during bounded model checking using R' than during bounded model checking using R. In practice, this can translate to a problem being solved within given time constraints, as opposed to timing out. Since transition relations of word-level RTL designs involve Boolean combinations of LMCs, *building an abstract transition relation requires existentially quantifying variables from Boolean combinations of LMCs*.

The above example illustrates the potential advantages of using an abstract transition relation obtained by existentially quantifying a subset of variables from the original transition relation. However, the effectiveness of this approach depends crucially on the choice of variables to quantify, on the availability of efficient techniques to obtain a quantifier-free version of the abstract transition relation, and on the amenability of the obtained abstract transition relation to efficient reasoning.

Let φ be a Boolean combination of LMCs over a set of variables V. We wish to compute a Boolean combination of LMCs ψ equivalent to $\exists X . \varphi$, where $X \subseteq V$. As we saw in Chapter 2, the problem of extending a QE algorithm for conjunctions of constraints to Boolean combinations of constraints is encountered in other first order theories such as linear real arithmetic and linear integer arithmetic as well. The techniques to solve this problem for these theories essentially transform the input Boolean combination of constraints to DNF and then apply the QE algorithm for conjunctions of constraints on each conjunction (monome) in the DNF. Section 2.4 gives a detailed survey of these techniques. Among these, the work by Chaki et. al. in [28] makes use of decision diagrams to represent Boolean combinations of octagonal constraints, and proposes efficient QE techniques that work

on decision diagrams. The work by Monniaux in [29] proposes an SMT solving based approach for extending Fourier-Motzkin to arbitrary Boolean combinations of constraints in linear real arithmetic. Our work in this chapter is motivated by the ideas introduced in these works.

Contributions: We present approaches to extend *Project* to eliminate quantifiers from Boolean combinations of LMCs. We introduce a new decision diagram called Linear Modular Decision Diagram (LMDD) that represents Boolean combinations of LMCs, and present algorithms for QE from LMDDs. We then present an SMT solving based approach and a hybrid approach that tries to combine the strengths of the LMDD and SMT solving based approaches. Experiments demonstrate the effectiveness of our techniques and indicate that the LMDD and SMT solving based approaches are incomparable, whereas the hybrid approach inherits the strengths of both LMDD and SMT solving based approaches. The experiments also demonstrate the utility of these techniques in bounded model checking of word-level RTL designs.

4.1 Linear Modular Decision Diagrams

A Linear Modular Decision Diagram (LMDD) is a data structure which represents Boolean combinations of LMCs. They are BDDs[51] with nodes labeled with LMEs or LMIs.

Formally an LMDD is a Directed Acyclic Graph (DAG) where the vertex set contains two terminal nodes 0 and 1 with out-degree zero and a set of non-terminal nodes with out-degree two. Each non-terminal node u is labeled with an LME or LMI denoted as $P(u)$. The children of a non-terminal node u are denoted by $H(u)$

and $L(u)$, and the node is denoted by the triple $(P(u),H(u),L(u))$. The child $H(u)$ is called high child and the child $L(u)$ is called low child. The edge set contains edges $(u,H(u))$ and $(u,L(u))$ for every non-terminal node u. An LMDD with root node u represents a formula $F(u)$ defined as $F(0) = \mathsf{false}$, $F(1) = \mathsf{true}$, $F(u) = \mathsf{ite}(P(u),F(H(u)),F(L(u)))$, where $\mathsf{ite}(\alpha,\beta,\gamma)$ denotes $(\alpha \wedge \beta) \vee (\neg\alpha \wedge \gamma)$. To simplify the notation, we will not distinguish between node u and the formula $F(u)$ represented by it.

Example: Fig. 4.3 shows an LMDD corresponding to the formula $\mathsf{ite}(x \leq y \pmod 4)$, $m = 2n+7 \pmod 8$, $m \neq 3n+5 \pmod 8)$. Note that the thick lines and dotted lines represent the edges $(u,H(u))$ and $(u,L(u))$ respectively for each non-terminal node u.

We define Reduced Ordered LMDDs similar to the way Reduced Ordered BDDs are defined. Let $\preceq$ be an ordering on the LMCs labeling the nodes of an LMDD f. The LMDD f is ordered w.r.t $\preceq$ if for every pair of non-terminal nodes u, v in f such that v is a child of u, we have $P(u) \preceq P(v)$. An LMDD f is ordered if there exists some ordering $\preceq$ on the LMCs labeling the nodes of f such that f is ordered w.r.t. $\preceq$. For example, the LMDD in Fig. 4.3 is ordered w.r.t. the order $x \leq y \pmod 4 \preceq m = 2n+7 \pmod 8 \preceq m = 3n+5 \pmod 8$. An LMDD is reduced iff (i) there are no duplicate nodes and (ii) there are no redundant nodes (a redundant node is a non-terminal node u such that $H(u) = L(u)$). The LMDD in Fig. 4.3 is a reduced LMDD. Hereafter we use LMDD to refer to Reduced Ordered LMDD.

The procedures for performing the basic operations on LMDDs and for construction of LMDDs are simple extensions of the corresponding procedures for BDDs.

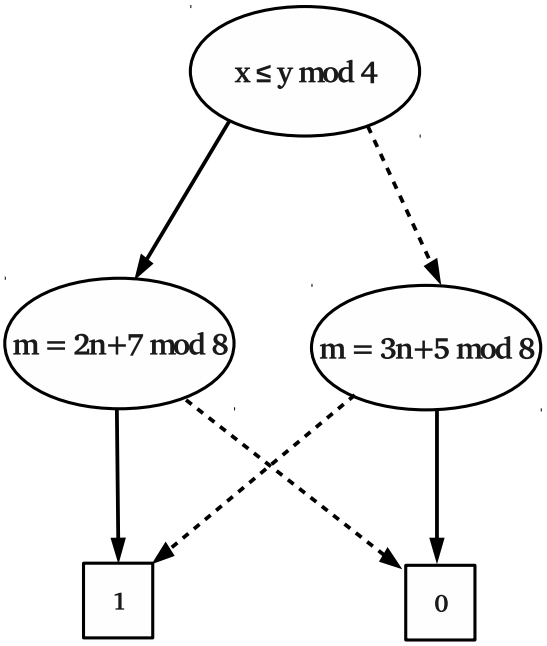

Figure 4.3: Example of an LMDD

Given LMDDs f and g, the function *Apply*(*op*, f, g) constructs an LMDD for f *op* g, where *op* is a binary operation. The implementation of *Apply* is similar to the implementation of the corresponding operation on BDDs. The case where either f or g is a terminal is straightforward. Consider the case where both f and g are non-terminals. If $P(f)$ and $P(g)$ are the same, then f *op* g is constructed as $(P(f), H(f)$ *op* $H(g), L(f)$ *op* $L(g))$. Otherwise if $P(f) \preceq P(g)$ then, f *op* g is constructed as $(P(f), H(f)$ *op* $g, L(f)$ *op* $g)$. Similarly if $P(g) \preceq P(f)$ then, f *op* g is constructed as $(P(g), f$ *op* $H(g), f$ *op* $L(g))$.

Given LMDDs f and g, the functions *AND* and *OR* construct LMDDs for $f \wedge g$ and $f \vee g$ respectively. They are implemented using *Apply*. The function *NOT*

that negates an LMDD f is implemented as *Apply*($\oplus$, f, 1), where $\oplus$ denotes the exclusive-or operation. Given LMDDs f and g, and an LMC c, the function *ITE* constructs an LMDD for $\mathrm{ite}(c, f, g)$. The implementation of *ITE* is similar to that of *Apply*. The function *createLMDD* constructs an LMDD from a Boolean combination of LMCs φ. Initially the LMDDs for the individual LMCs in φ are constructed. Then LMDD for φ is constructed recursively from the LMDDs for its sub-formulas using *Apply*.

4.1.1 Quantifier Elimination from LMDDs

The problem we wish to solve in this subsection can be formally stated as follows. Given an LMDD f representing a Boolean combination of LMCs over a set of variables V, we wish to compute an LMDD g equivalent to $\exists X.\, f$, where $X \subseteq V$.

The algorithms presented in this subsection use the following helper functions: i) *Vars*: returns the set of variables in an LMC, ii) *getConjunct*: computes the conjunction of LMCs in a given set, and iii) *isUnsat*: determines if the conjunction of LMCs in a given set is unsatisfiable.

A straightforward algorithm to compute $\exists X.\, f$ is to apply *Project* to each path originating from the root of f. We call this algorithm *All_Path_QElim* (see Algorithm 3). To compute $\exists X.\, f$, we call *All_Path_QElim* with arguments f, $\{\}$ and X. *All_Path_QElim* performs a recursive traversal of f collecting the set of LMCs S containing any of the variables in X that it encountered along the path from the root of f. If the path leads to a 1-terminal and if the conjunction C_s of LMCs in S is theory-consistent, then *Project* is called to compute $\exists X.\, C_s$.

Example: Consider the problem of computing $\exists X.\, f$, where f is the LMDD in Fig. 4.4 and $X = \{x\}$. As all LMCs in this example have modulus 8, we will

Algorithm 3: *All_Path_QElim*

Input: LMDD f, Set of LMCs S, Set of variables to eliminate X

Output: LMDD for $\exists X.(f \wedge C_s)$, where C_s is the conjunction of LMCs in S

```
1  if f = 0 or isUnsat(S) then
2      return 0;
3  if f = 1 then                    // f is theory-consistent 1-terminal
4      C_s := getConjunct(S);
5      π := Project(C_s, X);// π ≡ ∃X.C_s
6      return createLMDD(π);// π ≡ ∃X.(f ∧ C_s)
   // traverse down
7  c := P(f);
8  if Vars(c) ∩ X == { } then // c is free of variables to eliminate
9      return ITE(c, All_Path_QElim(H(f), S, X), All_Path_QElim(L(f), S,
       X));
10 else                             // c contains variables to eliminate
11     return OR(All_Path_QElim(H(f), S ∪ {c}, X), All_Path_QElim(L(f), S
       ∪ {¬c}, X));
```

not specifically write " $(\mathrm{mod}\ 8)$" for brevity. Note that there are two paths in f leading to 1-terminal with theory-consistent context: (i) $(3x+2y=0) \rightarrow (4x+m \leq 2) \rightarrow (2x+m=0)$ and (ii) $(3x+2y \neq 0) \rightarrow (2x+n=0)$. *All_Path_QElim* reduces $\exists X.f$ into the disjunction of (i) $\exists x.((3x+2y=0) \wedge (4x+m \leq 2) \wedge (2x+m=0))$ and (ii) $\exists x.((3x+2y \neq 0) \wedge (2x+n=0))$. *Project* computes $\exists x.((3x+2y=0) \wedge (4x+m \leq 2) \wedge (2x+m=0))$ as $(m \leq 2) \wedge (4y+m=0)$ and $\exists x.((3x+2y \neq 0) \wedge (2x+n=0))$ as $(4n=0)$. Thus the final result is LMDD for

$((m \leq 2) \wedge (4y + m = 0)) \vee (4n = 0)$.

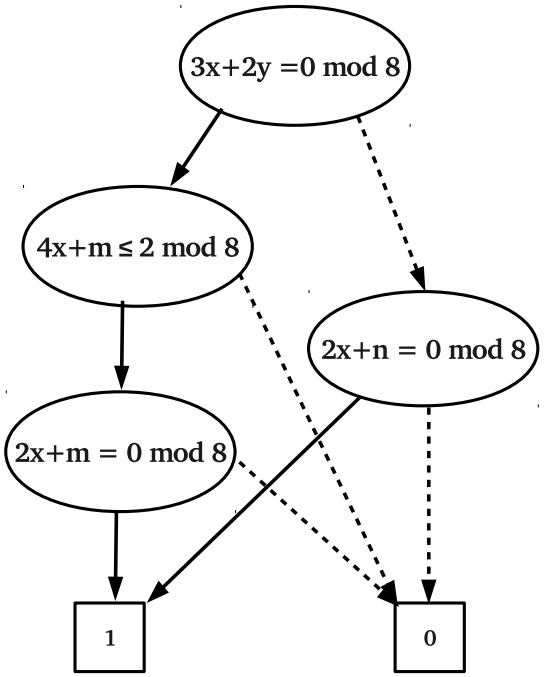

Figure 4.4: Example LMDD to illustrate QE

Recall that *All_Path_QElim* is similar to the algorithm proposed in Cavada et al's work in [27] (see Subsection 2.4.2). As observed in [28], because of the dependence of the result of a recursive call on the context S, if the same LMDD node is encountered following two different paths, then the results of the calls are not the same in general. Hence *All_Path_QElim* is not amenable to dynamic programming usually employed in the implementation of decision diagram operations. The number of recursive calls is linear in the number of paths in f, which can be exponential in the number of nodes in f.

In the following discussion we present a more efficient algorithm *QE_LMDD*

to compute $\exists X.\, f$. *QE_LMDD* makes use of an algorithm called *QE1_LMDD* that eliminates a single variable x from f (see Algorithm 4). To compute $\exists x.\, f$, we call *QE1_LMDD* with arguments f, $\{\}$ and x. *QE1_LMDD* performs a recursive traversal of the LMDD f collecting the set of LMCs S_x containing x that it encountered along the path from f.

In general, *QE1_LMDD* (f, S_x, x) computes an LMDD for $\exists x.\,(f \wedge C_{S_x})$, where C_{S_x} denotes the conjunction of LMCs in S_x. Let E_x be the set of LMEs in S_x. Let each LME e_i in E_x be of the form $2^{k_i} \cdot x = t_i$, where $k_i = \kappa(x, e_i)$ and $1 \leq i \leq n$ (recall the definition of κ from Section 3.1). Without loss of generality, let k_1 be the smallest among $k_1, \ldots, k_n$. Let g be any internal non-terminal node of f represented as $(P(g), H(g), L(g))$. Let us denote $P(g)$ by c. It can be observed that if c has x in its support, then c can be simplified by replacing the occurrences of $2^{k_1} \cdot x$ in it by t_1. Let c' be the simplified LMC. Note that if $\kappa(x, c) \geq k_1$, then c' we get, is free of x. Thus, if $\kappa(x, c) \geq k_1$, then g can be simplified to $(c', H(g), L(g))$, where c' is free of x.

We call the procedure that performs the selection of LME with the minimum κ among the LMEs in E_x as *selectLME*. The Procedure *simplifyLMDD* (see Algorithm 5) performs simplification of f using the selected LME as described above. The procedure *simplifyLMC* in Algorithm 5 simplifies c to c' using the selected LME.

If *simplifyLMDD* is successful in eliminating all occurrences of variable x using the selected LME, then *simplifyLMDD* returns a simplified LMDD f' such that $\exists x.\,(f \wedge C_{S_x})$ is equivalent to $f' \wedge \exists x.\,(C_{S_x})$. Note that $\exists x.\,(C_{S_x})$ can be computed by *Project*. In this case, *QE1_LMDD* returns without any further recursive calls. If *simplifyLMDD* is unable to eliminate all occurrences of variable x, then

Algorithm 4: *QE1_LMDD*

Input: LMDD f, Set of LMCs S_x, Variable to eliminate x

Output: LMDD for $\exists x.(f \wedge C_{S_x})$,

where C_{S_x} is the conjunction of LMCs in S_x

if $f = 0$ *or* isUnsat(S_x) **then**
  return 0;
if $f = 1$ **then** `// theory-consistent 1-terminal`
  $C_{S_x} := getConjunct(S_x)$;
  $\pi := Project(C_{S_x}, \{x\})$;// $\pi \equiv \exists x.C_{S_x}$
  return $createLMDD(\pi)$;// $\pi \equiv \exists x.(f \wedge C_{S_x})$

`// simplification using LMEs`

$E_x :=$ set of LMEs in S_x;
if $E_x \neq \{\}$ **then**
  $e_1 := selectLME(E_x)$;
  $f' := simplifyLMDD(f, e_1, x)$;
  if f' *is free of* x **then**
    $C_{S_x} := getConjunct(S_x)$;
    $\pi := Project(C_{S_x}, \{x\})$;// $\pi \equiv \exists x.C_{S_x}$
    return $AND(f', createLMDD(\pi))$;// $f' \wedge \pi \equiv \exists x.(f \wedge C_{S_x})$
else
  $f' := f$;

`// traverse down`

$c := P(f')$;
if c *is free of* x **then**
  return $ITE(c, QE1\_LMDD(H(f'), S_x, x), QE1\_LMDD(L(f'), S_x, x))$;
else
  return $OR(QE1\_LMDD(H(f'), S_x \cup \{c\}, x), QE1\_LMDD(L(f'), S \cup \{\neg c\}, x))$;

Algorithm 5: *simplifyLMDD*

Input: LMDD f, LME $e_1 : 2^{k_1} \cdot x = t_1$, Variable to eliminate x

Output: LMDD f simplified using e_1

```
1 if f = 0 or f = 1 then
2     return f;
3 c := P(f);
4 if c is free of x then
5     return ITE(c, simplifyLMDD(H(f), x, e_1), simplifyLMDD(L(f), x,
      e_1));
6 else
7     c' := simplifyLMC(c, e_1, x);// if κ(x,c) ≥ k_1, then c' is free
          of x
8     return ITE(c', simplifyLMDD(H(f), x, e_1), simplifyLMDD(L(f), x,
      e_1));
```

QE1_LMDD proceeds by recursively traversing the simplified LMDD f'.

Example: Let us understand how *QE1_LMDD* computes $\exists x.\, f$, where f is the LMDD in Fig. 4.4. *QE1_LMDD* calls *simplifyLMDD* with arguments $H(f)$, $(3x+2y=0)$ and x. Note that the LME $(3x+2y=0)$ is equivalent to $(x=2y)$ modulo 8. *simplifyLMDD* eliminates all occurrences of x in $H(f)$ using $(x=2y)$, and thus simplifies $H(f)$ as shown in Fig. 4.5. Let g be the simplified LMDD, which is free of x (shown in different colour in Fig. 4.5). Notice that $\exists x.\,(H(f) \wedge (x=2y))$ is equivalent to $g \wedge \exists x.\,(x=2y)$. Since $\exists x.\,(x=2y)$ is true, $\exists x.\,(H(f) \wedge (x=2y))$ is equivalent to g. However, $L(f)$ cannot be simplified in this manner, as there are no LMEs involving x in its context. *QE1_LMDD* performs traversal of $L(f)$, and calls *Project* to compute $\exists x.\,((3x+2y \neq 0) \wedge (2x+n=0))$. *Project* computes $\exists x.\,((3x+2y \neq 0) \wedge (2x+n=0))$ as $(4n=0)$. Thus the final result is LMDD for $g \vee (4n=0)$.

It can be observed that if the same LMDD node is encountered with the same LME following two different paths, then the results of the calls to *simplifyLMDD* must be the same. Hence *simplifyLMDD* can be implemented with dynamic programming. Moreover, although the result of each recursive call to *QE1_LMDD* depends on the context S_x, the number of LMCs in S_x is usually very small, as only the LMCs containing x are collected in S_x. Hence *QE1_LMDD* is still amenable to dynamic programming.

QE1_LMDD can be repeatedly invoked to compute $\exists X.\, f$. This is implemented in the algorithm *QE_LMDD*. The order in which variables are selected for elimination in *QE_LMDD* has a crucial impact on the sizes of the intermediate and final LMDDs. In our ordering scheme, we selected the variable occurring in the least number of LMDD nodes as the next variable to be eliminated. Intu-

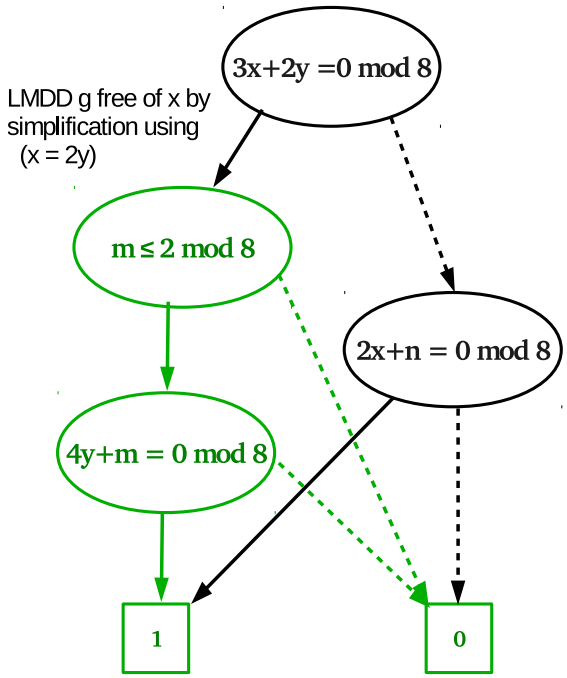

Figure 4.5: Example for *simplifyLMDD*

itively, this ordering scheme usually results in smaller contexts (i.e., smaller S_x's), and more opportunities for dynamic programming.

In practice, the strategy of eliminating one variable at a time and simplification of LMDDs using the LMEs in the context provide significant opportunities for reuse of results through dynamic programming. As a result of these, *QE_LMDD* in practice clearly outperforms *All_Path_QElim*, as also demonstrated by our experiments.

4.2 QE using SMT Solving

In recent years there has been significant progress in SMT solvers for bit-vector arithmetic. In this section we present an approach for QE from Boolean combinations of LMCs that leverages progress in SMT solvers for bit-vector arithmetic. Given a Boolean combination of LMCs φ over a set of variables V, we wish to compute a Boolean combination of LMCs ψ equivalent to $\exists X.\varphi$, where $X \subseteq V$.

We initially present an algorithm *All_SMT* (see Algorithm 6) to compute $\exists X.\varphi$, which is a straightforward extension of the All-SMT loop given in Section 2.4. Initially the satisfiability of φ is checked using an SMT solver. If φ is unsatisfiable, then $\exists X.\varphi$ is false. Otherwise, the solution m_1 of φ obtained from the solver is generalized to a conjunction of LMCs C_1 such that $C_1 \Rightarrow \varphi$. The SMT solver is now called to check if $\varphi \wedge \neg C_1$ is satisfiable. If $\varphi \wedge \neg C_1$ is unsatisfiable, then $\exists X.\varphi$ is equivalent to $\exists X.C_1$. Otherwise, the solution m_2 of $\varphi \wedge \neg C_1$ obtained is generalized to a conjunction of LMCs C_2 such that $C_2 \Rightarrow \varphi$. This loop is repeated until the formula given to the SMT solver becomes unsatisfiable. Each iteration i of the loop generates a conjunction of LMCs C_i such that $C_i \Rightarrow \varphi$, for $1 \leq i \leq n$. Finally $\exists X.\varphi$ is equivalent to $\exists X.C_1 \vee \cdots \vee \exists X.C_n$. *Project* is used to compute $\exists X.C_i$, for $1 \leq i \leq n$.

Generalize1 (see Algorithm 7) uses the method suggested in [29] for generalizing a solution m of φ to a conjunction of LMCs C such that $C \Rightarrow \varphi$. *Generalize1* computes C as follows. First C is initialized to true. Each LMC c in φ is then evaluated with values given to variables in its support as per m. If c evaluates to true under m, i.e., $m \models c$, then c is conjoined with C. Otherwise, if c evaluates to false under m, i.e., $m \models \neg c$, then $\neg c$ is conjoined with C. It is easy to see that the conjunction C returned implies φ.

Algorithm 6: *All_SMT*

Input: Boolean combination of LMCs φ, Set of variables to eliminate X

Output: Boolean combination of LMCs ψ equivalent to $\exists X.\varphi$

```
1 H := φ;
2 ψ := false;
3 while H is satisfiable do
4     m := a solution of H;// m ⊨ H and m ⊨ φ
5     C := Generalize1(φ, m);// C ⇒ φ
6     π := Project(C, X);// π ≡ ∃X.C
7     ψ := ψ ∨ π;
8     H := H ∧ ¬C;
9 return ψ;// ψ ≡ ∃X.φ
```

Algorithm 7: *Generalize1*

Input: Boolean combination of LMCs φ, A solution m of φ

Output: A conjunction C of LMCs such that $C \Rightarrow \varphi$

```
1 S := set of LMCs in φ;
2 C := true;
3 for c ∈ S do
4     if m ⊨ c then
5         C := C ∧ c;
6     else
7         C := C ∧ ¬c;
8 return C;
```

Let us understand the working of *All_SMT* with an example. We will use this as a running example throughout this section.

Example: Consider the problem of computing $\exists X.\varphi$, where φ is $(y = 4x) \wedge((x \neq z) \vee (x \neq w))$ with modulus of all LMCs as 8, and $X = \{x\}$. Let $m_1 : x = 1, y = 4, z = 0, w = 0$ be the solution of φ from the SMT solver. Note that *Generalize1* generalizes m_1 to the conjunction $C_1 : (y = 4x) \wedge (x \neq z) \wedge (x \neq w)$, and *Project* computes $\exists x.C_1$ as $\pi_1 : (2y = 0)$. As $\varphi \wedge \neg C_1$ is satisfiable, the next iteration of the loop starts. Let $m_2 : x = 1, y = 4, z = 1, w = 0$ be the solution of $\varphi \wedge \neg C_1$ from the SMT solver. The conjunction C_2 from *Generalize1* obtained by generalizing m_2 is $(y = 4x) \wedge (x = z) \wedge (x \neq w)$, and $\pi_2 \equiv \exists x.C_2$ is $(y = 4z) \wedge (z \neq w)$. Note that $\varphi \wedge \neg C_1 \wedge \neg C_2$ is satisfiable, and the third iteration starts. Let solution m_3 of $\varphi \wedge \neg C_1 \wedge \neg C_2$ be $x = 1, y = 4, z = 0, w = 1$. The conjunction C_3 from *Generalize1* is $(y = 4x) \wedge (x \neq z) \wedge (x = w)$, and $\pi_3 \equiv \exists x.C_3$ is $(y = 4w) \wedge (z \neq w)$. The loop now terminates as $\varphi \wedge \neg C_1 \wedge \neg C_2 \wedge \neg C_3$ is unsatisfiable. The result of QE is $\pi_1 \vee \pi_2 \vee \pi_3$, i.e., $(2y = 0) \vee ((y = 4z) \wedge (z \neq w)) \vee ((y = 4w) \wedge (z \neq w))$.

As mentioned in Section 2.4, the work by Monniaux in [29] improves the All-SMT loop in the following ways.

1. Instead of $\neg C$, $\neg\exists X.C$ is conjoined with the formula H, checked for satisfiability. This is called "interleaving projection and model enumeration". It is observed that this enables pruning the solution space of the problem, which results in early termination of the algorithm.

2. Before computing $\exists X.C$, C is generalized by dropping unnecessary constraints that do not affect the validity of $C \Rightarrow \varphi$. Generalizing C by dropping unnecessary constraints simplifies C, and reduces the time to compute $\exists X.C$. Moreover, it results in generalized $\exists X.C$, which increases the size of

solution space pruned by conjoining $\neg\exists X.C$ with H.

Algorithm 8: *QE_SMT*

Input: Boolean combination of LMCs φ, Set of variables to eliminate X

Output: Boolean combination of LMCs ψ equivalent to $\exists X.\varphi$

```
1  H := φ;
2  ψ := false;
3  while H is satisfiable do
4      m := a solution of H;// m ⊨ H and m ⊨ φ
5      C := Generalize1(φ, m);// C ⇒ φ
6      C' := Generalize2(φ, C);// C ⇒ C' and C' ⇒ φ
7      π := Project(C', X);// π ≡ ∃X.C'
8      ψ := ψ ∨ π;
9      H := H ∧ ¬π;
10 return ψ;// ψ ≡ ∃X.φ
```

Our algorithm *QE_SMT* (see Algorithm 8) makes use of these optimizations to compute $\exists X.\varphi$. The algorithm *QE_SMT* calls the procedure *Generalize2* for generalizing C by dropping unnecessary constraints from C. Thus C' computed by *Generalize2* is such that $C \Rightarrow C'$ and $C' \Rightarrow \varphi$. The implementation of *Generalize2* in [29] works as follows. For each constraint c in C, it is checked to see if $C \Rightarrow \varphi$ remains valid even after dropping c from C. If $C \Rightarrow \varphi$ remains valid even after dropping c from C, then c is unnecessary and is dropped from C. Otherwise if the implication $C \Rightarrow \varphi$ becomes invalid after dropping c from C, then c is not dropped from C. Checking the validity of $C \Rightarrow \varphi$ involves an SMT solver call. However, in our experiments with LMCs, we have found that this implementation

of *Generalize2* is prohibitively time consuming as the number of SMT solver calls is equal to the number of constraints in C. Hence our implementation of *Generalize2* makes use of a cheaper technique to achieve generalization.

Our technique is based on analysis of the Boolean skeleton of the formula φ. Boolean skeleton P of φ is the representation of Boolean structure of φ as a Directed Acyclic Graph (DAG), with leaves representing LMCs in φ and internal nodes as $\neg$, $\wedge$, and $\vee$. As every LMC in φ appears in C in its original or negated form, C effectively gives an assignment of Boolean values to the leaves of P. We now perform a bottom-up traversal of P to evaluate P using the values assigned to the leaves. Let $B(n)$ be the value assigned to a node n in P during the evaluation. For each node n, we find a subset $S(n)$ of LMCs in C that are sufficient to evaluate n to $B(n)$. Table 4.1 shows how $B(n)$ and $S(n)$ are computed for the different nodes in P under different conditions. Let $S(r)$ be the set of LMCs found in this way for the root r of P. C' is computed as the conjunction of LMCs in $S(r)$. It is easy to see that $C \Rightarrow C'$ and $C' \Rightarrow \varphi$.

Example: In our example, where φ is $(y = 4x) \wedge((x \neq z) \vee (x \neq w))$, consider the case when C is $(y = 4x) \wedge(x = z) \wedge (x \neq w)$. The Boolean skeleton P of φ is $n_1 \wedge (n_2 \vee n_3)$, where n_1, n_2, n_3 denote $(y = 4x)$, $(x \neq z)$, $(x \neq w)$ respectively. Note that $B(n_1) = \mathsf{true}$, $B(n_2) = \mathsf{false}$, and $B(n_3) = \mathsf{true}$. Also $S(n_1) = \{n_1\}$, $S(n_2) = \{\neg n_2\}$, and $S(n_3) = \{n_3\}$. Let n_4 be the node $(n_2 \vee n_3)$. Since $B(n_2) = \mathsf{false}$, $B(n_3) = \mathsf{true}$, and n_4 is $(n_2 \vee n_3)$, we have $B(n_4) = \mathsf{true}$. Note that $B(n_3) = \mathsf{true}$ is sufficient to make $B(n_4) = \mathsf{true}$. Hence $S(n_4) = S(n_3) = \{n_3\}$ as per Table 4.1. Let r be the root node of P, i.e., the node $n_1 \wedge n_4$. Since $B(n_1) = \mathsf{true}$, $B(n_4) = \mathsf{true}$, we have $B(r) = \mathsf{true}$. Since r is $n_1 \wedge n_4$, both $B(n_1)$ and $B(n_4)$ should be true for $B(r)$ to be true. Hence we have $S(r) = S(n_1) \cup S(n_4) = \{n_1, n_3\}$. Therefore C' is

Table 4.1: Computation of $B(n)$ and $S(n)$ inside *Generalize2*

node n	**Condition**	$B(n)$	$S(n)$
LMC c	c appears in C	$true$	$\{c\}$
	$\neg c$ appears in C	$false$	$\{\neg c\}$
$\neg n_1$	$B(n_1) = true$	$false$	$S(n_1)$
	$B(n_1) = false$	$true$	$S(n_1)$
$n_1 \wedge n_2$	$B(n_1) = true \wedge B(n_2) = true$	$true$	$S(n_1) \cup S(n_2)$
	$B(n_1) = true \wedge B(n_2) = false$	$false$	$S(n_2)$
	$B(n_1) = false \wedge B(n_2) = true$	$false$	$S(n_1)$
	$B(n_1) = false \wedge B(n_2) = false$	$false$	smaller among $S(n_1)$ and $S(n_2)$
$n_1 \vee n_2$	$B(n_1) = true \wedge B(n_2) = true$	$true$	smaller among $S(n_1)$ and $S(n_2)$
	$B(n_1) = true \wedge B(n_2) = false$	$true$	$S(n_1)$
	$B(n_1) = false \wedge B(n_2) = true$	$true$	$S(n_2)$
	$B(n_1) = false \wedge B(n_2) = false$	$false$	$S(n_1) \cup S(n_2)$

$n_1 \wedge n_3$, i.e., $(y = 4x) \wedge (x \neq w)$.

Let us understand the working of *QE_SMT* on this example. Let $m : x = 1$, $y = 4$, $z = 1$, $w = 0$ be the solution of φ from the SMT solver in the first iteration. Note that *Generalize1* generalizes m to the conjunction $C : (y = 4x) \wedge (x = z) \wedge (x \neq w)$. As we just saw, *Generalize2* generalizes C to $C' : (y = 4x) \wedge (x \neq w)$. *Project* computes $\exists x.C'$ as $\pi : (2y = 0)$. Note that $\varphi \wedge \neg\pi$ is unsatisfiable, and the algorithm terminates. The result of QE is π, i.e., $(2y = 0)$.

Note that the optimizations in *QE_SMT* helped us in early termination of the loop in this example. *All_SMT* had taken 3 iterations, whereas *QE_SMT* finished in just 1 iteration. In practice, these optimizations provide significant improvement in performance, as we will see in Section 4.4.

4.3 Hybrid Approach

The factors that contribute to the success of the LMDD-based approach are the presence of large shared sub-LMDDs and the strategy of eliminating one variable at a time. Both factors contribute to significant opportunities for reuse of results through dynamic programming. The success of the SMT-based approach is attributable primarily to pruning of the solution space achieved by interleaving of projection and model enumeration. In the following discussion, we present a hybrid approach that tries to combine the strengths of these two approaches.

We illustrate the idea with the help of an example.

Example: Consider the working of *QE_LMDD* on the example of computing $\exists X.\varphi$, where φ is $(y = 4x) \wedge ((x \neq z) \vee (x \neq w))$ with modulus of all LMCs as 8, and $X = \{x\}$. Fig. 4.6 shows LMDD for φ with order $(y = 4x) \preceq (x = z) \preceq (x = w)$.

The LMDD nodes are denoted as f_1, f_2, f_3, f_4, and f_5. Recall that *QE_LMDD*

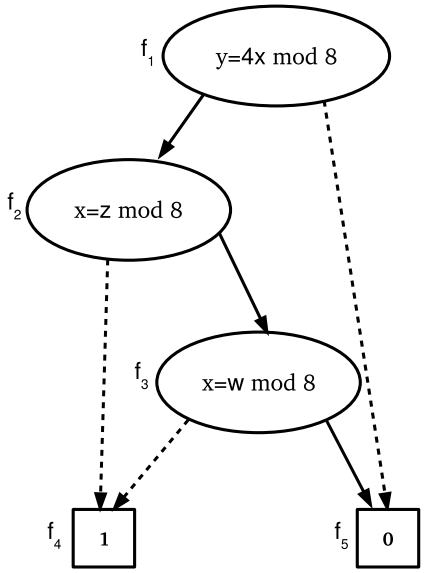

Figure 4.6: Example for hybrid approach

calls *QE1_LMDD* with arguments f_1, $\{\}$ and x to compute $\exists x.\varphi$. *QE1_LMDD*(f_1, $\{\}$, x) makes recursive calls *QE1_LMDD*(f_2, $\{y = 4x\}$, x) and *QE1_LMDD*(f_5, $\{y \neq 4x\}$, x). *QE1_LMDD*(f_2, $\{y = 4x\}$, x) makes further recursive calls *QE1_LMDD*(f_4, $\{y = 4x,\ x \neq z\}$, x) and *QE1_LMDD*(f_3, $\{y = 4x,\ x = z\}$, x).

QE1_LMDD(f_4, $\{y = 4x,\ x \neq z\}$, x) calls *Project* to compute $\exists x.((y = 4x) \wedge (x \neq z))$, and returns LMDD for $(2y = 0)$. *QE1_LMDD*(f_3, $\{y = 4x,\ x = z\}$, x) returns LMDD for $(y = 4z) \wedge \mathsf{ite}(z = w, 0, 1)$, and *QE1_LMDD*(f_5, $\{y \neq 4x\}$, x) returns 0. Thus the result of QE is LMDD for $(2y = 0) \vee ((y = 4z) \wedge \mathsf{ite}(z = w, 0, 1))$. Note that effectively *QE1_LMDD* splits $\exists x.\varphi$ into three sub-problems: (i) $\exists x.(f_4 \wedge (y = 4x) \wedge (x \neq z))$, (ii) $\exists x.(f_3 \wedge (y = 4x) \wedge (x = z))$, and (iii) $\exists x.(f_5 \wedge (y \neq 4x))$. The result of QE is the disjunction of the results of these sub-problems.

Note that $\exists x.\varphi$ is actually equivalent to $(2y = 0)$, the result of the first sub-problem $\exists x.(f_4 \wedge (y = 4x) \wedge (x \neq z))$. Hence we could have avoided the computation of the sub-problems $\exists x.(f_3 \wedge (y = 4x) \wedge (x = z))$ and $\exists x.(f_5 \wedge (y \neq 4x))$. We call such sub-problems whose computation can be avoided as "redundant" sub-problems. We can infer that the sub-problems $\exists x.(f_3 \wedge (y = 4x) \wedge (x = z))$ and $\exists x.(f_5 \wedge (y \neq 4x))$ are redundant, from the fact that $f_3 \wedge (y = 4x) \wedge (x = z) \wedge (2y \neq 0)$ and $f_5 \wedge (y \neq 4x) \wedge (2y \neq 0)$ are unsatisfiable.

In general, suppose we wish to compute $\exists X.f$, where f denotes an LMDD representing a Boolean combination of LMCs over a set of variables V and $X \subseteq V$. We can derive a set of sub-problems of the form $\exists X.(f_i \wedge C_i)$, for $1 \leq i \leq n$, where f_i denotes an LMDD and C_i denotes a conjunction of LMCs, such that $\exists X.f$ is equivalent to $\bigvee_{i=1}^{n}(\exists X.(f_i \wedge C_i))$. Let g denote $\bigvee_{i=1}^{m}(\exists X.(f_i \wedge C_i))$, where $1 \leq m < n$. A sub-problem $\exists X.(f_j \wedge C_j)$, where $m+1 \leq j \leq n$, is redundant if $f_j \wedge C_j \wedge \neg g$ is unsatisfiable.

Our hybrid algorithm *QE_Combined* (see Algorithm 9) makes use of this idea to identify redundant sub-problems. Initially *QE_Combined* selects a satisfiable path π in the LMDD f using a function *selectPath*. Subsequently, the algorithm *simplify* (see Algorithm 10) is invoked, which traverses the path π, in order to split f into an equivalent disjunction $\bigvee_{i=1}^{n}(f_i \wedge C_i)$, where f_i denotes an LMDD and C_i denotes a conjunction of LMCs. $(f_i \wedge C_i)$ is represented in Algorithm 10 as a pair $\langle f_i, C_i \rangle$.

In order to split LMDD f, *simplify* is called with arguments f, π, C all initialized to true, and S initialized to $\{\}$. *simplify* collects $(f_i \wedge C_i)$, for $1 \leq i \leq n$ in the set S in the following way. The path π is traversed recursively starting from the root node of f, conjoining with C all LMCs encountered on π. In each recursive

Algorithm 9: *QE_Combined*

Input: LMDD f, Set of variables to eliminate X

Output: Boolean combination of LMCs g equivalent to $\exists X.\, f$

```
1  π := selectPath(f);
2  S := { };// set of sub-problems
3  C := true;
4  simplify(f, π, C, S);
5  g := false;
6  for each ⟨f_i, C_i⟩ ∈ S do
7      if f_i ∧ C_i ∧ ¬g is satisfiable then
8          h := QE_LMDD_Context(f_i, C_i, X);
9          g := g ∨ h;
10 return g;
```

call, if f is a terminal, then $\langle f, C\rangle$ is inserted in S. Otherwise if f is a non-terminal and node $H(f)$ appears in π, then $\langle L(f), C \wedge \neg P(f)\rangle$ is inserted in S. Similarly if f is a non-terminal and node $L(f)$ appears in π, then $\langle H(f), C \wedge P(f)\rangle$ is inserted in S. Fig. 4.7 illustrates the splitting scheme followed by *simplify*.

Example (Continued): In the case of LMDD in Fig. 4.6, using the path $(y = 4x) \rightarrow (x \neq z) \rightarrow 1$ as π splits the LMDD into (i) $\langle f_4,\ (y = 4x) \wedge (x \neq z)\rangle$, (ii) $\langle f_3,\ (y = 4x) \wedge (x = z)\rangle$, and (iii) $\langle f_5,\ (y \neq 4x)\rangle$.

The function *selectPath* selects the path π in the following way. First, a solution m of f is generated using an SMT solver call. The root node of f is selected as the first node in π. The LMC $P(f)$ labeling the root node of f is then evaluated with values given to variables in its support as per m. If $P(f)$ evaluates to true

Algorithm 10: *Simplify*

Input: LMDD f, Satisfiable path π,
Conjunction C of LMCs encountered along π

Output: Set of sub-problems S

```
1 if f = 1 then
2     S := S ∪ { ⟨f, C⟩ };
3 else
4     if node H(f) is in π then
5         S := S ∪ { ⟨L(f), C∧¬P(f)⟩ };
6         simplify(H(f), π, C∧P(f));
7     else
8         S := S ∪ { ⟨H(f), C∧P(f)⟩ };
9         simplify(L(f), π, C∧¬P(f));
```

under m, then $H(f)$ is selected as the next node in π. Otherwise if $P(f)$ evaluates to false under m, then $L(f)$ is selected as the next node in π. The LMC labeling the child of f thus selected as the next node in π is then evaluated under m. These steps are iteratively repeated until 1-terminal is encountered, each iteration adding a new node to π. Note that encountering 1-terminal is guaranteed since m is a solution of f.

QE_Combined now computes $g \equiv \exists X.\, f$ as $\bigvee_{i=1}^{n} (\exists X.\, (f_i \wedge C_i))$ in the following manner. In order to compute $\exists X.\, (f_i \wedge C_i)$, *QE_Combined* makes use of an algorithm *QE_LMDD_Context*. *QE_LMDD_Context* is a variant of *QE_LMDD* that eliminates a set of variables from an LMDD conjoined with a set of LMCs.

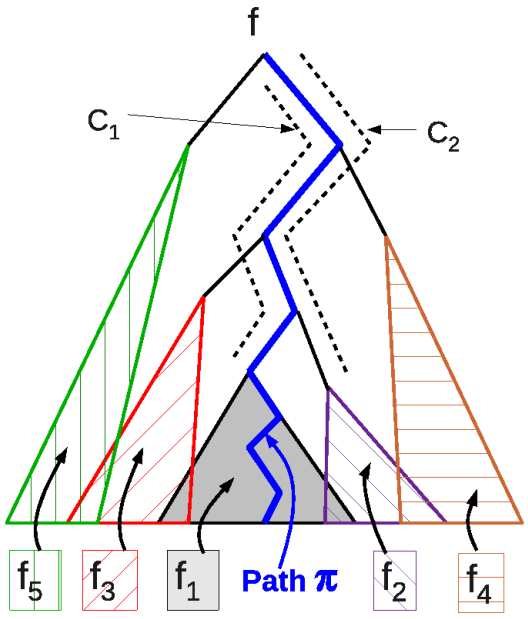

Figure 4.7: Deriving $f_i \wedge C_i$ from path π

QE_Combined initially sets g to false. In the first iteration of the loop, the satisfiability of $f_1 \wedge C_1$ is checked. If $f_1 \wedge C_1$ is satisfiable, then g is set to $\exists X.(f_1 \wedge C_1)$. Otherwise if $f_1 \wedge C_1$ is unsatisfiable, then the sub-problem $\exists X.(f_1 \wedge C_1)$ is redundant and is not computed. In the second iteration, the satisfiability of $f_2 \wedge C_2 \wedge \neg g$ is checked. If $f_2 \wedge C_2 \wedge \neg g$ is satisfiable, then $\exists X.(f_2 \wedge C_2)$ is computed and is disjoined with g. Otherwise if $f_2 \wedge C_2 \wedge \neg g$ is unsatisfiable, then $\exists X.(f_2 \wedge C_2)$ is not computed. This loop is repeated until all the sub-problems are considered. It can be observed that g is equivalent to $\bigvee_{j=1}^{i}\left(\exists X.(f_j \wedge C_j)\right)$ after the i^{th} iteration of the loop, which implies that g is equivalent to $\bigvee_{j=1}^{n}\left(\exists X.(f_j \wedge C_j)\right)$ when the loop is terminated.

Example (Continued): In our example, in the first iteration of the loop, the satisfiability of $f_4 \wedge (y = 4x) \wedge (x \neq z)$ is checked. Since $f_4 \wedge (y = 4x) \wedge (x \neq z)$

is satisfiable, g is set to $(2y = 0)$, the result of $\exists x.(f_4 \wedge (y = 4x) \wedge (x \neq z))$. In the second iteration, the satisfiability of $f_3 \wedge (y = 4x) \wedge (x = z) \wedge (2y \neq 0)$ is checked. Since $f_3 \wedge (y = 4x) \wedge (x = z) \wedge (2y \neq 0)$ is unsatisfiable, $\exists x.(f_3 \wedge (y = 4x) \wedge (x = z))$ is not computed. Similarly, in the third iteration of the loop, the satisfiability of $f_5 \wedge (y \neq 4x) \wedge (2y \neq 0)$ is checked. Note that $f_5 \wedge (y \neq 4x) \wedge (2y \neq 0)$ is unsatisfiable. Hence $\exists x.(f_5 \wedge (y \neq 4x))$ is also not computed. The final result of QE is $(2y = 0)$.

Note that unlike *QE_SMT*, *QE_Combined* does not explicitly interleave projections inside model enumeration. However disjoining the result of $\exists X.(f_i \wedge C_i)$ with g, and computing $\exists X.(f_i \wedge C_i)$ only if $f_i \wedge C_i \wedge \neg g$ is satisfiable, helps in avoiding the computation of redundant sub-problems. This enables pruning the solution space of the problem, as achieved in *QE_SMT*.

4.4 Experimental Results

We performed experimental evaluation of our QE techniques in three different ways. First we performed experiments to evaluate the performance and effectiveness of *QE_LMDD*, *QE_SMT*, and *QE_Combined*. We then compared the performance of *QE_SMT* with alternative QE techniques based on bit-blasting and conversion to linear integer arithmetic. Finally we performed experiments to evaluate the utility of our QE techniques in verification.

The experiments were performed on a 1.83 GHz Intel(R) Core 2 Duo machine with 2GB memory running Linux, with a timeout of 1800 seconds. We implemented our own LMDD package for carrying out QE experiments involving LMDDs. In all our experiments, we used simplifyingSTP [102] as the SMT

solver. We selected simplifyingSTP, because it has a variable eliminator [79] which is considered as suitable for solving bit-vector formulas involving LMEs. In all experiments using LMDDs, we used the same ordering on LMCs labeling the LMDD nodes. We performed depth-first traversal of the DAG representations of formulas from which the LMDDs were created; LMCs were ordered in the order they were encountered in the depth-first traversal.

Simplification heuristics: We used the following simplification heuristics in our implementation.

1. The LMDs with modulus 2 were converted to equivalent LMEs. For example, the LMD $x+y \neq 1 \pmod 2$ was converted to $x+y=0 \pmod 2$. We observed that this helps in easy elimination of existentially quantified variables involved in LMCs with modulus 2.

2. In a non-terminal LMDD node u, if $P(u)$ is an LME, then it is kept in a normal form $2^k \cdot x = t$, where x is the variable appearing first in lexicographical ordering between the names of variables in the support of $P(u)$, and $k = \kappa(x, P(u))$ (recall the definition of κ from Section 3.1). This allows identification of equivalent LMEs during LMDD creation and hence more compact LMDDs.

Evaluation of *QE_SMT*, *QE_LMDD*, and *QE_Combined*: We used the same benchmark suite consisting of 198 *lindd* benchmarks and 39 *vhdl* benchmarks, that we used for experiments in Chapter 3. Each of these benchmarks is a Boolean combination of LMCs with a subset of the variables in their support existentially quantified. The details of these benchmarks can be found in Section 3.6. As mentioned in Section 3.6, the total number of variables, the number of variables

to be eliminated, and the number of bits to be eliminated in the *lindd* benchmarks ranged from 30 to 259, 23 to 207, and 368 to 3312 respectively. The total number of variables, the number of variables to be eliminated, and the number of bits to be eliminated in the *vhdl* benchmarks ranged from 8 to 50, 2 to 21, and 10 to 672 respectively.

We measured the time taken by *QE_SMT*, *QE_LMDD*, and *QE_Combined* for QE from each benchmark. For *QE_LMDD* and *QE_Combined*, this included the time to build the initial LMDD. We observed that each approach performed better than the others for some benchmarks (see Fig. 4.8 and Fig. 4.9). Note that the points in Fig. 4.9 are scattered, while the points in Fig. 4.8(a) and 4.8(b) are more clustered near the 45° line. This shows that *DD* and *SMT* based approaches are incomparable, whereas the hybrid approach inherits the strengths of both *DD* and *SMT* based approaches. Hence, given a problem instance, we recommend the hybrid approach, unless the approach which will perform better is known a-priori.

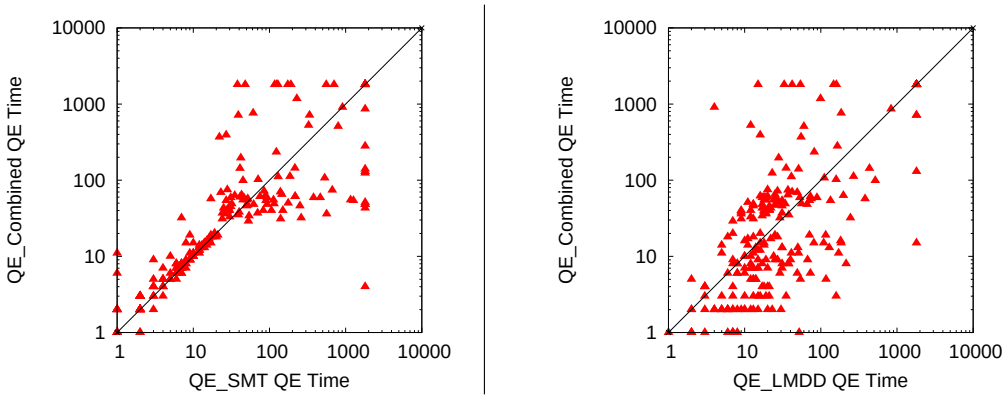

Figure 4.8: Plots comparing (a) *QE_SMT* and *QE_Combined* and (b) *QE_LMDD* and *QE_Combined* (All times are in seconds)

In order to evaluate the effectiveness of our simplifications in *QE_LMDD*, we

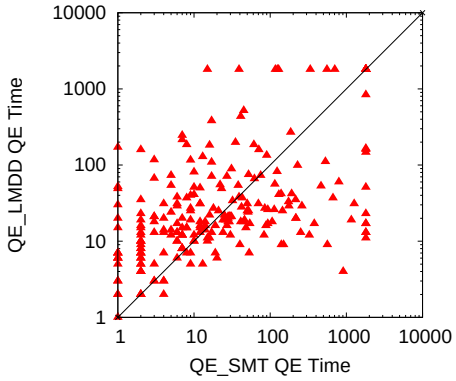

Figure 4.9: Plot comparing *QE_SMT* and *QE_LMDD* (All times are in seconds)

compared the time taken by *QE_LMDD* with that taken by *All_Path_QElim* for QE from each benchmark (see Fig. 4.10(a)). *All_Path_QElim* succeeded only in a few cases. This is not surprising, as the LMDDs for the benchmarks contained a huge number of paths. In *QE_LMDD*, the single variable elimination strategy and the simplification of LMDDs using *simplifyLMDD* helped in achieving significant reuse of results through dynamic programming. This helped in avoiding path enumeration, which resulted in considerable performance gains over *All_Path_QElim*.

In order to evaluate the effectiveness of our generalization technique based on analysis of Boolean skeleton of formulae in *Generalize2*, we implemented a variant of *QE_SMT* called *QE_SMT_Mod*. *QE_SMT_Mod* is the same as *QE_SMT* except that it uses the implementation of *Generalize2* as proposed in [29]. Recall from Subsection 4.2 that the implementation of *Generalize2* in [29] makes use of SMT solver calls to identify unnecessary LMCs. We compared the time taken by *QE_SMT* and *QE_SMT_Mod* for QE from each benchmark (see Fig. 4.10(b)). *QE_SMT* outperformed *QE_SMT_Mod* except in a few cases. On profiling, we found that most of the time taken by *QE_SMT_Mod* was spent in the SMT solver calls in *Generalize2*. In the few cases where *QE_SMT_Mod* performed better than

Table 4.2: Comparison between QE algorithms

Benchmark	D	T	V	B	AQ	QL	AS	QS_M	QS
i_choose_cb.smt	299	42	33	528	8	8	TO	50	1
i_cb_print.smt	428	61	48	768	390	6	TO	764	18
i_gnt_text_view_clicked.smt	589	86	68	1088	414	15	TO	836	11
i_check_for_ext_cb.smt	378	50	39	624	1230	12	TO	188	5
i_connect2Server_with_af.smt	427	54	43	688	TO	14	TO	226	6
i_context_menu.smt	757	102	81	1296	TO	37	TO	439	5
i_bmedit_action.smt	607	83	66	1056	TO	12	TO	361	5
i_SoundBeep.smt	516	72	57	912	TO	18	TO	1337	23
i_action_movePage.smt	702	104	83	1328	TO	24	TO	437	8
i_add_face.smt	531	73	58	928	TO	12	TO	104	3
i_avi_check_file.smt	660	113	90	1440	TO	17	TO	684	13
i_Cudd_EquivDC.smt	681	101	80	1280	TO	54	TO	1718	21
i_gnt_tree_clicked.smt	658	87	69	1104	TO	21	TO	TO	24
i_Cudd_bddLeqUnless.smt	783	106	84	1344	TO	53	TO	TO	37
i_check_idleness.smt	704	91	72	1152	TO	36	TO	TO	42
i_addTriangleRecur.smt	776	116	92	1472	TO	24	TO	TO	49
i_command_subst_completion_function.smt	627	100	79	1264	TO	16	TO	TO	53
i_gnt_tree_size_changed.smt	752	107	85	1360	TO	24	TO	TO	20
i_bash_quote_filename.smt	640	92	73	1168	TO	18	TO	TO	45
i_cb_message.smt	544	79	63	1008	TO	30	TO	TO	69

All times are in seconds. **TO**: $>$ 1800 seconds, **D**: Dag size of the formula, **T**: Total number of variables, **V**: Number of variables to be eliminated, **B**: Number of bits to be eliminated, **AQ**: Total time taken by *All_Path_QElim*, **QL**: Total time taken by *QE_LMDD*, **AS**: Total time taken by *All_SMT*, **QS_M**: Total time taken by *QE_SMT_Mod*, **QS**: Total time taken by *QE_SMT*

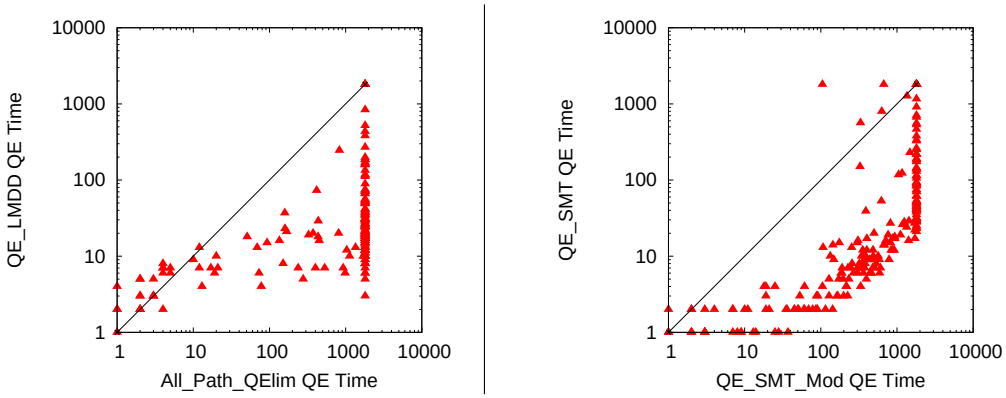

Figure 4.10: Plots comparing (a) *All_Path_QElim* and *QE_LMDD* and (b) *QE_SMT* and *QE_SMT_Mod* (All times are in seconds)

QE_SMT, the SMT solver based generalization in *QE_SMT_Mod* was more effective which helped in faster termination of the All-SMT loop. The comparison indicates that our generalization technique based on analysis of the Boolean skeleton of formulae in *Generalize2* is a cheaper and effective alternative to the SMT solver based generalization technique in [29]. Table 4.2 gives the comparison between the times taken by the different QE algorithms for a sample of 20 *lindd* benchmarks.

Recall that in *QE_Combined*, we converted $\exists X.\, f$, where f is an LMDD, into an equivalent disjunction of sub-problems, and then gave these sub-problems to *QE_LMDD_Context* separately. Our analysis revealed that this helped in identifying redundant sub-problems. However, it was observed that splitting $\exists X.\, f$ into sub-problems and computing the sub-problems separately, reduced scope for reuse of results through dynamic programming when compared to computing $\exists X.\, f$ directly using *QE_LMDD*. We could also observe that using a more eager strategy for splitting into subproblems (i.e., a strategy that generates more sub-

problems) in place of *simplify*, further reduced scope for reuse of results, although it improved opportunity for identifying redundant sub-problems. On the other hand, using a less eager strategy improved reuse of results, but gave lesser opportunity for identifying redundant sub-problems. Hence, although reuse of results and splitting into subproblems contribute towards success of the hybrid approach, they act against each other. In our experiments, we found that the splitting scheme in *simplify* achieves a trade-off between them.

Comparison with Alternative QE Techniques: We wanted to understand how *QE_SMT* would perform if a bit-blasting or linear integer arithmetic based alternative QE algorithm is used in place of *Project*. In order to do this, we first computed the average times taken by *Project* for QE from conjunction-level problem instances arising from *QE_SMT* when QE is performed on each benchmark. We also computed the average times taken by *Layer1_Blast*, *Layer1_OT*, and *Layer2_OT* (see Section 3.6) for QE from these conjunction-level problem instances. For each benchmark, we then compared the average QE times taken by *Project* against those taken by *Layer1_Blast* and *Layer1_OT* (see Fig. 4.11(a) and 4.11(b)). Subsequently, for each benchmark, we compared the average time consumed by *Layer3* in the *Project* calls with that consumed by Omega Test in the *Layer2_OT* calls (see Fig. 4.12). For a large number of benchmarks, we observed that the bit-blasting or linear integer arithmetic based alternative QE algorithm was unsuccessful in eliminating quantifiers from the conjunction-level problem instances. These benchmarks are indicated by the topmost green circles in Fig. 4.11(a), Fig. 4.11(b), and Fig. 4.12. Note that, for these benchmarks we could not compute the average times consumed by the bit-blasting or linear integer arithmetic based alternative QE algorithm, as the algorithm was unsuccessful in

eliminating quantifiers from the conjunction-level problem instances. There were a few cases where Omega Test performed better than *Layer3*. This was due to the relatively larger number of recursive *Project* calls in these cases.

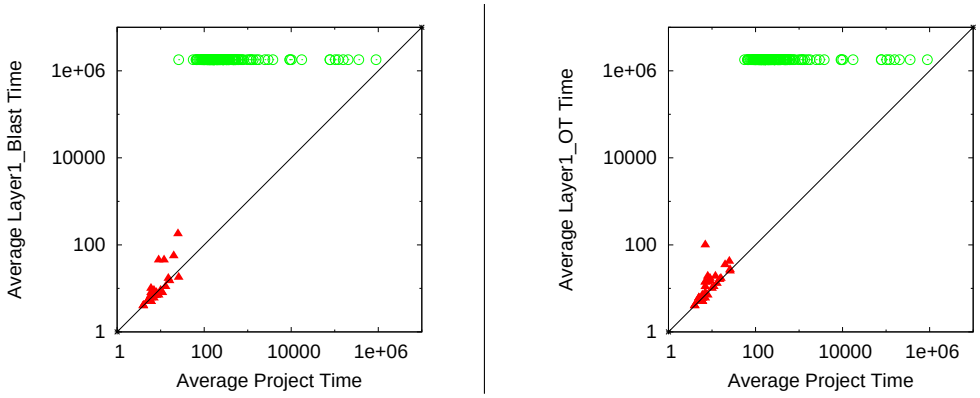

Figure 4.11: Plots comparing average times consumed by (a) *Project* and *Layer1_Blast* and (b) *Project* and *Layer1_OT* when used inside *QE_SMT* (All times are in milliseconds). Topmost green circles indicate the benchmarks for which *Layer1_Blast* or *Layer1_OT* was unsuccessful.

We also wanted to understand how *QE_SMT* would perform if the BDD based alternative technique *BddBasedLayer2* (see Section 3.6) is used in place of *Layer2* inside *Project*. In order to do this, for each benchmark, we first computed the average time consumed by *Layer2* when QE is performed using *QE_SMT*. For each benchmark, we then computed the average time consumed by *BddBasedLayer2* when *BddBasedLayer2* is used in place of *Layer2* inside *Project*. Fig. 4.13(a) compares these times. Many points corresponding to different benchmarks are merged in Fig. 4.13(a), since the average times consumed in *Layer2* were significantly small compared those consumed in *BddBasedLayer2*. We provide a comparison of the total times in Fig. 4.13(b) for better exposition. The plots clearly

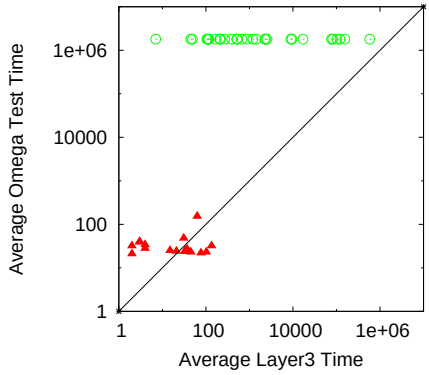

Figure 4.12: Plot comparing average times consumed by *Layer3* and Omega Test when used inside *QE_SMT* (All times are in milliseconds). Topmost green circles indicate the benchmarks for which Omega Test was unsuccessful.

demonstrate that *QE_SMT* performs poorly when the BDD based alternative technique is used in place of *Layer2*. Note that here again, topmost green circles in Fig. 4.13(a) and Fig. 4.13(b) indicate the benchmarks for which QE was unsuccessful when *BddBasedLayer2* was used in place of *Layer2*.

Utility of our QE algorithms in verification: Recall from Section 3.6 that the *vhdl* benchmarks were obtained by quantifying out a subset of internal variables from the symbolic transition relations of word-level VHDL designs. The quantifier eliminated formulae give abstract transition relations of the VHDL designs. In order to evaluate the utility of our QE algorithms, we used *QE_LMDD* to compute these abstract transition relations, and then used these abstract transition relations for checking safety properties of the VHDL designs using bounded model checking.

In order to check if the safety property holds for the first N cycles of operation, we first unrolled the transition relation N times, and conjoined the unrolled relation with the negation of the property. The resulting formula was then given to

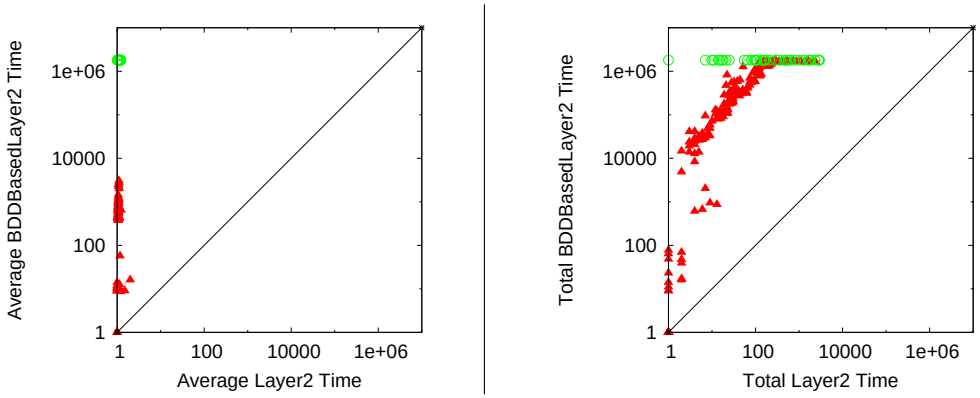

Figure 4.13: Plot comparing (a) average times and (b) total times consumed by *Layer2* and *BddBasedLayer2* when used inside *QE_SMT* (All times are in milliseconds). Topmost green circles indicate the benchmarks for which *BddBasedLayer2* was unsuccessful.

an SMT solver for checking satisfiability. Next, we obtained an abstract transition relation R' using *QE_LMDD*. The abstract transition relation was then unrolled N times and was conjoined with the negation of the property to obtain a formula, which was given to the SMT solver to check satisfiability.

All the SMT solver calls were unsatisfiable, which implies that the properties hold for the first N cycles of operation of the designs, and the abstract transition relations are sufficient to prove the properties. Table 4.3 gives a summary of the results for 16 designs. machine_1 to machine_12 are modified versions of benchmarks from ITC99 benchmark suite [69]. The remaining designs are proprietary. The table clearly shows the significant performance benefit of using abstract transition relations computed by *QE_LMDD* in these verification exercises.

For all the designs except machine_12, all the internal variables were eliminated from the transition relation in order to obtain the abstract transition relation.

Table 4.3: Experimental results on VHDL programs

Design	**LOC**	**TR**	**N=500**	
			NA	**QL**
machine_1	363	(592, 22, 580)	TO(TO)	52(7, 23)
machine_2	373	(594, 22, 436)	TO(TO)	30(6, 1)
machine_3	383	(620, 25, 439)	TO(TO)	33(6, 3)
machine_4	253	(439, 26, 677)	1471(1441)	24(2, 0)
machine_5	253	(439, 26, 509)	1443(1413)	25(2, 0)
machine_6	363	(406, 17, 64)	78(53)	17(1, 1)
machine_7	379	(440, 22, 69)	221(196)	22(1, 3)
machine_8	251	(286, 20, 157)	193(177)	13(2, 0)
machine_9	251	(286, 20, 485)	331(315)	13(2, 0)
machine_10	363	(406, 17, 420)	TO(TO)	16(0, 1)
machine_11	363	(593, 22, 96)	TO(TO)	40(8, 4)
machine_12	363	(406, 17, 420)	TO(TO)	220(4, 187)
board_1	404	(400, 24, 194)	1442(1424)	21(12, 1)
board_2	373	(420, 24, 194)	TO(TO)	14(5, 1)
board_3	503	(573, 54, 361)	TO(TO)	16(5, 1)
board_4	415	(422, 28, 198)	241(223)	62(9, 2)

All times are in seconds. **TO**: $>$ 1800 seconds, **LOC**: Lines of code, **TR**: Transition relation details (dag size, number of variables, number of bits), **NA**: Without abstraction : total time (simplifyingSTP time), **QL**: With *QE_LMDD* for abstraction : total time (*QE_LMDD* time, simplifyingSTP time), **N**: Number of BMC unrollings

For machine_12, a manually chosen subset of internal variables were eliminated. It was observed that in all the cases, *Layer1* and *Layer2* were sufficient to eliminate the variables, without any call to *Layer3*. *Layer2* was needed only in five cases: machine_6 through machine_10. In these cases *Layer2* eliminated 12.5% to 40% of the quantified variables.

We performed limited preliminary experiments to evaluate the utility of *Layer1* and *Layer2* as preprocessing steps for conjunctions of LMCs before finding satisfying assignments for the conjunctions using an SMT solver. Towards this end, we generated 9 sets of random benchmarks. Each set included 5 benchmarks that are randomly generated conjunctions of LMCs with the same number of variables, LMEs, LMDs and LMIs. The moduli of all LMCs in all benchmarks was fixed to 2^{24}. The number of variables varied from 20 to 50. The number of LMCs was chosen as twice the number of variables.

In order to properly evaluate the effectiveness of *Layer1* and *Layer2*, we generated three types of benchmarks. Type-1 benchmarks contained an equal mix of LMEs, LMDs, and LMIs. The benchmarks in set_1, set_4, and set_7 in Table 4.4 were of this type. These benchmarks allowed us to evaluate the effectiveness of *Layer1*. In type-2 benchmarks, 80% of constraints were LMDs and the remaining were LMIs. The benchmarks in set_2, set_5, and set_8 in Table 4.4 were of this type. Finally, in type-3 benchmarks, 80% of constraints were LMIs and the remaining were LMDs. The benchmarks in set_3, set_6, and set_9 in Table 4.4 were of type-3. Type-2 and type-3 benchmarks allowed us to evaluate the effectiveness of *Layer2* on different mixes of constraints.

We first measured the time taken by simplifyingSTP to solve each benchmark. We then eliminated variables in the support of each benchmark using *Layer1* and

Layer2. This yields a potentially simplified benchmark with fewer variables in the support. We then measured the time taken by simplifyingSTP to solve each preprocessed benchmark. Table 4.4 gives a summary of the results. Preprocessing helped in cases of type-2 benchmark sets set_2, set_5, and set_8. Preprocessing in these cases completely solved the problem instances. In other cases preprocessing either caused additional overhead or was of not much use.

Table 4.4: Experimental results on preprocessing using *Layer1* and *Layer2*

Set	**V**	**E**	**D**	**I**	**NP**	**PR**	**AP**
set_1	20	14	13	13	1763	1572	2688
set_2	20	0	36	4	3270	251	0
set_3	20	0	4	36	3208	655	3245
set_4	30	20	20	20	8415	4769	9216
set_5	30	0	54	6	7423	533	0
set_6	30	0	6	54	7203	1651	7218
set_7	40	28	26	26	223880	11255	171207
set_8	40	0	72	8	14115	1150	0
set_9	40	0	8	72	14343	3561	13238

All times are in milliseconds. **V**: Number of variables, **E**: Number of LMEs, **D**: Number of LMDs, **I**: Number of LMIs, **NP**: Average time taken by simplifyingSTP for solving the benchmarks in the set without preprocessing, **PR**: Average time for preprocessing the benchmarks in the set, **AP**: Average time taken by simplifyingSTP for solving the benchmarks in the set after preprocessing

We also performed limited preliminary experiments to evaluate the utility of

our QE techniques for computing Craig interpolants for Boolean combinations of LMCs. Towards this end, we generated a set of interpolation benchmarks. Each benchmark is a pair of formulas (φ, ψ), where φ, ψ are Boolean combinations of LMCs which are mutually inconsistent. We denote the set of variables in the support of both φ and ψ as Y. The set of variables in the support of φ but not in the support of ψ is denoted as X. Similarly, the set of variables in the support of ψ but not in the support of φ is denoted as Z.

Note that $\exists X.\varphi$ serves as an interpolant for (φ, ψ). In fact, $\exists X.\varphi$ is the *strongest* interpolant for (φ, ψ). For each interpolation benchmark, we first used *QE_Combined* to compute $\exists X.\varphi$. For each benchmark, we then used Mathsat to compute an interpolant (Mathsat makes use of work in [67] for interpolant computation). We then compared the time taken by Mathsat to compute interpolant with that taken by *QE_Combined* to compute $\exists X.\varphi$. Table 4.5 gives a summary of the results for 10 benchmarks. Since interpolation is an approximation of QE, it is amenable to simplifications that QE may not be able exploit. Nevertheless our experiments show that the two techniques are incomparable. In some cases, an interpolant can be computed faster than the quantifier-eliminated formula, while in other cases, QE using our techniques can be done much faster than computing an interpolant using the techniques encoded in MathSAT.

Considering the three sets of experiments that we performed for evaluating the utility of our QE techniques, it can be seen that our techniques are convincingly useful for computing abstract transition relations in bounded model checking. Our experiments showed that applying our techniques often translates to a model checking problem being solved within given time constraints, as opposed to timing out. However the other two sets of experiments – applying our tech-

Table 4.5: Experimental results on computing interpolants

Benchmark	$\|\mathbf{X}\|$	$\|\mathbf{Y}\|$	$\|\mathbf{Z}\|$	**W**	**MS**	**QC**
benchmark_1	5	16	12	16	12	36
benchmark_2	6	15	8	16	45	44
benchmark_3	8	10	6	8	0	3
benchmark_4	17	23	11	32	7	275
benchmark_5	17	23	10	32	6	142
benchmark_6	21	25	8	32	TO	TO
benchmark_7	12	7	7	22	TO	16
benchmark_8	10	17	8	32	0	29
benchmark_9	3	10	3	32	TO	12
benchmark_10	4	14	3	16	TO	7

All times are in seconds. **TO**: $>$ 1800 seconds, $|\mathbf{X}|$: Number of variables in set X, $|\mathbf{Y}|$: Number of variables in set Y, $|\mathbf{Z}|$: Number of variables in set Z, **W**: Maximum bit-width of a variable, **MS**: Time taken by Mathsat, **QC**: Time taken by *QE_Combined*

niques in solving conjunctions of LMCs and for computing Craig interpolants for Boolean combinations of LMCs – gave mixed results. Exploring other applications of our techniques is part of future work.

4.5 Conclusions

Extending QE algorithms that work on conjunctions on constraints to eliminate quantifiers from arbitrary Boolean combinations of constraints is an important problem. In this chapter, we presented three approaches for extending a QE algorithm for conjunctions of LMCs to eliminate quantifiers from Boolean combinations of LMCs. Our experiments indicated that the LMDD and SMT solving based approaches are incomparable, and our hybrid approach inherits the strengths of both LMDD and SMT solving based approaches. The experiments also demonstrated the effectiveness of our QE approaches and their utility in computing abstract transition relations in bounded model checking of word-level RTL designs. Our approaches clearly made the difference between successful and timed out verification runs. In the next chapter, we will focus on QE from propositional logic.

Chapter 5

Quantifier Elimination for Propositional Formulas

In this chapter, we focus on techniques for QE from propositional formulas that are based on Skolem functions. Skolem functions, introduced by Thoraf Skolem in the 1920s, have long occupied a central role in mathematical logic. Formally, given a first-order logic formula $F(x,y)$, a *Skolem function* for x in F is a function $\psi(y)$ such that substituting $\psi(y)$ for x in F yields a formula equivalent to $\exists x.F(x,y)$, i.e. $F(\psi(y),y) \equiv \exists x.F(x,y)$. Classically, Skolem functions have been used to prove fundamental theorems in logic. More recently, with the advent of fast SAT solvers and theorem provers, several practically relevant problems have been encoded as quantified formulas, and can be solved by generating Skolem functions.

We focus on the case where the formula F is propositional. It follows from the definition of Skolem function that QE can be achieved by substituting Skolem functions for existentially quantified variables. Other than QE, Skolem functions

have many important applications. Skolem functions are used as certificates [8] for satisfiable Quantified Boolean Formulas (QBFs) by QBF solvers. The problem of synthesizing a circuit or program [9] that satisfies the specification $Spec(I,O)$, where I is the set of inputs and O is the set of outputs reduces to computing Skolem functions $\psi(I)$ for variables in O in the formula $Spec(I,O)$.

Motivating Application: As mentioned in Chapter 1, our primary motivation for studying Skolem function generation comes from the problem of computing disjunctive decompositions of sequential circuits represented as symbolic transition functions [6]. The disjunctive decomposition problem asks the following: Given a sequential circuit, can we obtain "component" sequential circuits, each of which has the same state space as the original circuit, but only a single transition going out of every state such that the set of state transitions of the original circuit is the union of the sets of state transitions of the components ? We illustrate the disjunctive decomposition problem with the help of an example.

Consider a sequential circuit as shown in Figure 5.1. The circuit consists of a combinational logic block and a set of flip-flops (denoted as F/F). The circuit has state variables y_1, y_2 and a single input x. The state transition behaviour of the circuit is specified by its symbolic transition function, shown inside the combinational block, where y_1' and y_2' refer to next state versions of y_1 and y_2 respectively. The state transition diagram of the circuit is shown on the right in Figure 5.1. Each state in the state transition diagram is labeled by a valuation of the state variables, and each edge is labeled by the valuation of the input variable that enables the corresponding state transition.

Two components of this circuit are shown in Figure 5.2 and Figure 5.3, along

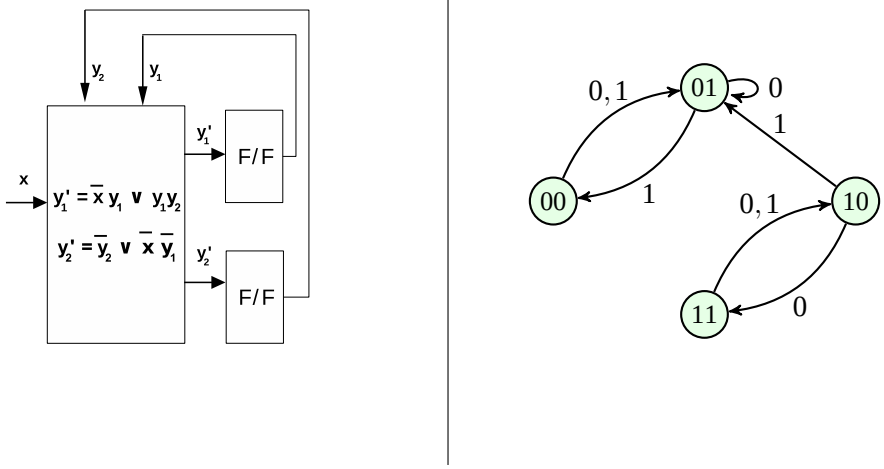

Figure 5.1: An example sequential circuit

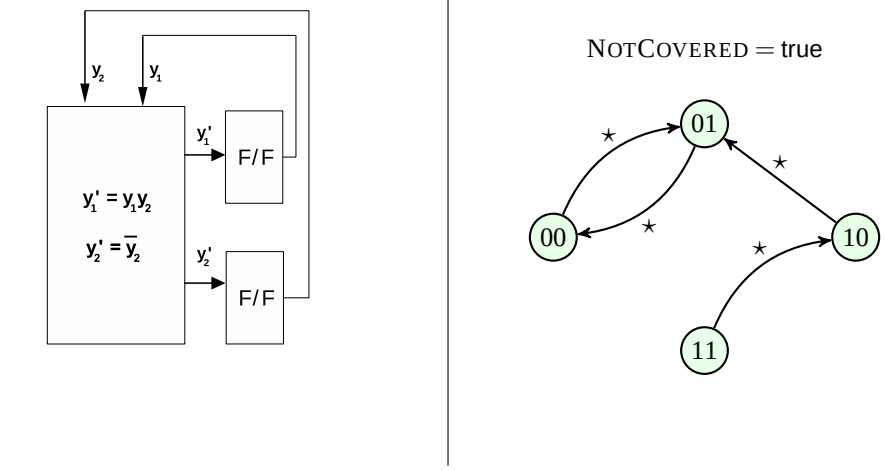

Figure 5.2: First component of example sequential circuit

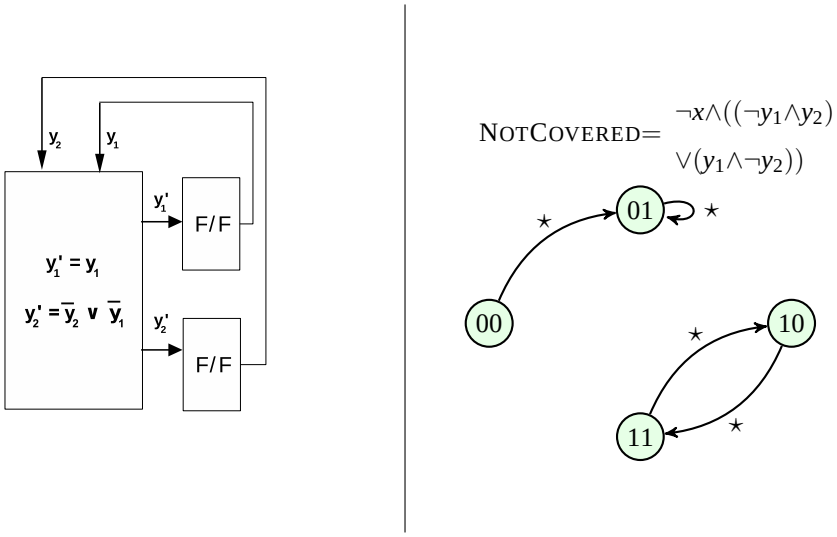

Figure 5.3: Second component of example sequential circuit

with their state transition diagrams. Note that all states in the components have single outgoing transitions and all state transitions of the original circuit are present in at least one of the two components.

The goal of disjunctive decomposition is to come up with symbolic transition functions for the components, given the symbolic transition function for the original sequential circuit. This problem can be trivially solved if the state transition diagram for the original circuit is constructed. However constructing the state transition diagram may not always be feasible for circuits with large numbers of state variables.

A naive approach to solve this problem is to substitute all possible values for the input variables in the symbolic transition function of the original sequential circuit one-by-one to generate all components. Let m be the number of input variables. The above approach would generate 2^m components. If there is a state in the state transition diagram of the circuit that has 2^m outgoing transitions, i.e., a state

that goes to different states for all possible valuations of input variables, then at least 2^m components are needed to compute the disjunctive decomposition. However, such a scenario arises very rarely in practice. Moreover, in state transition diagrams of circuits that arise in practice, often there are a lot of state transitions that are labeled by multiple valuations of input variables. We can exploit this feature to reduce the number of components generated.

Let X be the set of input variables and Y be the set of present state variables. Let NOTCOVERED(X, Y) be a propositional formula that captures the set of transitions that are not present in any of the components generated so far. Such transitions that are not present in any of the components generated so far are said to be transitions that are *not covered* by the components. For example in Fig. 5.3 after generating the first component, the transitions corresponding to input 0 from state 01, and input 0 from state 10 are not covered, as characterized by $\text{NOTCOVERED}(x, y_1, y_2) = \neg x \wedge ((\neg y_1 \wedge y_2) \vee (y_1 \wedge \neg y_2))$.

Generating a new component requires choosing values of input variables as functions of the present state such that for every state, the outgoing transition enabled by the chosen values of input variables is not covered by the components generated so far. If all outgoing transitions from a state are covered, then we can choose any of the outgoing transitions for that state.

It can be observed that the process of choosing values of input variables as functions of the present state as above, essentially reduces to finding functions $\psi(Y)$ such that $\forall Y.\,(\exists X.\,\text{NOTCOVERED}(X,Y) \Rightarrow \text{NOTCOVERED}(\psi(Y),Y))$ holds. Since $\forall Y.\ (\neg\exists X.\,\text{NOTCOVERED}(X,Y) \Rightarrow \neg\text{NOTCOVERED}(\psi(Y),Y))$ holds for any $\psi(Y)$, we are actually interested in computing $\psi(Y)$ such that $\exists X.\ \text{NOTCOVERED}\ (X,Y) \equiv \text{NOTCOVERED}\ (\psi(Y),Y)$. Recalling the definition of Skolem

functions, this is the same as generating Skolem functions for variables in X in the formula NOTCOVERED(X, Y).

For example in Fig. 5.2, initially NOTCOVERED(x,y_1,y_2) is true. Note that $(y_1 \wedge \neg y_2) \vee (\neg y_1 \wedge y_2)$ is a Skolem function for x in true. Selecting the outgoing transition enabled by choosing x as $(y_1 \wedge \neg y_2) \vee (\neg y_1 \wedge y_2)$ for every state gives us the first component. NOTCOVERED(x,y_1,y_2) after generating the first component is $\neg x \wedge ((\neg y_1 \wedge y_2) \vee (y_1 \wedge \neg y_2))$. Note that $(y_1 \wedge y_2) \vee (\neg y_1 \wedge \neg y_2)$ is a Skolem function for x in $\neg x \wedge ((\neg y_1 \wedge y_2) \vee (y_1 \wedge \neg y_2))$, and selecting the outgoing transition enabled by choosing x as $(y_1 \wedge y_2) \vee (\neg y_1 \wedge \neg y_2)$ for every state gives us the second component.

In computation of disjunctive decompositions, if all outgoing transitions from a state are already covered by the components generated so far, then $\exists X$. NOTCOVERED(X, Y) is not valid, although it can be satisfiable. For example, in Fig. 5.2, all outgoing transitions from state 00 as well as state 11 are covered by the first component. Hence $\exists X$. NOTCOVERED(X, Y) after generation of the first component, i.e., $\exists x.\ (\neg x \wedge ((\neg y_1 \wedge y_2) \vee (y_1 \wedge \neg y_2)))$ is not valid, although it is satisfiable. Therefore, in general, we need to find Skolem functions for variables in X in NOTCOVERED(X, Y) irrespective of the validity of $\exists X$. NOTCOVERED(X, Y).

Let $F(X,Y)$ be a propositional formula, where X and Y denote the sequences of variables $(x_1,\ldots x_n)$ and $(y_1,\ldots y_m)$, respectively. Suppose we are interested in generating Skolem functions for variables in X in $F(X,Y)$. Moreover, suppose $F(X,Y)$ is in Negation Normal Form. There are two interesting cases here: (i) $F(X,Y)$ is a disjunction of sub-formulas, and (ii) $F(X,Y)$ is a conjunction of sub-formulas.

The case where $F(X,Y)$ is a disjunction of sub-formulas is easy to solve. For example, let $F(X,Y)$ be of the form $F_1 \vee F_2$. Let ψ_1 be a Skolem function for x_i in F_1, and ψ_2 be a Skolem function for x_i in F_2. It can be observed that $((\exists x_i.F_1) \wedge \psi_1) \vee ((\neg\exists x_i.F_1) \wedge \psi_2)$ can be used as a Skolem function for x_i in F. Note that this avoids treating $F_1 \vee F_2$ as a single monolithic formula, and computes Skolem function for x_i in F from Skolem functions for x_i in F_1 and F_2.

The case where $F(X,Y)$ is a conjunction of sub-formulas is the harder case. Interestingly, for several problem instances, the specification of $F(X,Y)$ is available in such a *factored* form, i.e. as conjunction of simpler sub-formulas, each of which depends on a subset of variables appearing in the formula $F(X,Y)$. Existing algorithms for Skolem function generation ignore any such factored form and treat the conjunction of factors as a single monolithic function. We show that exploiting the factored form can yield significant performance advantages when generating Skolem functions.

We are not aware of other techniques for Skolem function generation that exploit the factored form of a formula. As mentioned in Section 2.5.2, earlier work on Skolem function generation essentially belong to one of four categories. The first category includes techniques that extract Skolem functions from a proof of validity of $\exists X.\ F(X,Y)$ [17, 16, 93]. In problem instances where $\exists X.\ F(X,Y)$ is valid (and it forms an important sub-class of problems), these techniques can usually find succinct Skolem functions if there exists a short proof of validity. However, in several other important classes of problems such as QE and computation of disjunctive decomposition, although the formula $\exists X.\ F(X,Y)$ is satisfiable, it may not be valid, and techniques in the first category cannot be applied.

The second category of techniques for Skolem function generation includes

techniques that use templates of candidate Skolem functions [9]. Template-based techniques are effective only when the set of candidate Skolem functions is known and small. While this is a reasonable assumption in some domains [9], it is not possible to identify a small set of candidate Skolem functions in other domains. BDD-based techniques [94] are yet another way to compute Skolem functions. Unfortunately, these techniques are known not to scale well, unless custom-crafted variable orders are used. The last category includes techniques that use cofactors to obtain Skolem functions [18, 6]. These techniques do not exploit the factored representation of a formula and, as we show experimentally, do not scale well to large problem instances.

Contributions: Our main technical contribution is a SAT-based Counter-Example Guided Abstraction-Refinement (CEGAR) algorithm for generating Skolem functions from factored formulas. Unlike competing approaches, our algorithm exploits the factored representation of a formula and leverages advances made in SAT-solving technology. The factored representation is used to arrive at an initial abstraction of Skolem functions, while a SAT-solver is used as an oracle to identify counter-examples that are used to refine the Skolem functions until no counter-examples exist. We present a detailed experimental evaluation of our algorithm over a large class of benchmarks. We also present experiments that compare performance of our algorithm vis-a-vis state-of-the-art algorithms [18, 17]. Our experiments show that on several large problem instances, we outperform competing algorithms both in terms of time and Skolem function size.

5.1 Preliminaries

We use lower case letters (possibly with subscripts) to denote propositional variables, and upper case letters to denote sequences of such variables. We use 0 and 1 to denote the propositional constants false and true, respectively. Let $F(X,Y)$ be a propositional formula, where X and Y denote the sequences of variables $(x_1,\ldots x_n)$ and $(y_1,\ldots y_m)$, respectively. We are interested in problem instances where $F(X,Y)$ is given as a conjunction of factors $f^1(X_1,Y_1),\ldots f^r(X_r,Y_r)$, where each X_i (resp., Y_i) is a possibly empty sub-sequence of X (resp., Y). For notational convenience, we use F and $\bigwedge_{i=1}^{r} f^i$ interchangeably throughout this chapter. The set of variables in F is called the *support* of F, and is denoted $\mathsf{Supp}(F)$. Given propositional formulas F and Ψ, we use $F[x_i \mapsto \Psi]$ to denote the formula obtained by substituting every occurrence of the variable x_i in F with Ψ. This is also conventionally called *function composition*. If X is a sequence of variables and x_i is a variable, we use $X \setminus x_i$ to denote the sub-sequence of X obtained by removing x_i (if present) from X. Abusing notation, we also use X to denote the set of elements in X, when there is no confusion.

Definition 1. *Given a propositional formula $F(X,Y)$, a* Skolem function *for $x_i \in X$ in $F(X,Y)$ is a function $\psi(X \setminus x_i, Y)$ such that $\exists x_i . F \equiv F[x_i \mapsto \psi]$.*

Example: Consider the propositional formula $x_1 \wedge y_1$. It can be observed that y_1 and 1 are two Skolem functions for the variable x_1 in $x_1 \wedge y_1$, since (i) $\exists x_1 . (x_1 \wedge y_1)$ is equivalent to y_1, (ii) $(x_1 \wedge y_1)[x_1 \mapsto y_1]$ is equivalent to y_1, and (iii) $(x_1 \wedge y_1)[x_1 \mapsto 1]$ is equivalent y_1.

Notice that a Skolem function for x_i in F need not be unique. The following proposition from [18, 6] characterizes the space of all Skolem functions for x_i in

F.

Proposition 8. *A function $\psi(X \setminus x_i, Y)$ is a Skolem function for x_i in $F(X,Y)$ iff $F[x_i \mapsto 1] \wedge \neg F[x_i \mapsto 0] \Rightarrow \psi$ and $\psi \Rightarrow F[x_i \mapsto 1] \vee \neg F[x_i \mapsto 0]$.*

Proof of Proposition 8. For any given value of variables in $X \setminus x_i \cup Y$, there are four possible cases:

1. $\neg F[x_i \mapsto 1] \wedge \neg F[x_i \mapsto 0]$: For this value of variables in $X \setminus x_i \cup Y$, F is 0 irrespective of the value of x_i. Hence irrespective of whether ψ is 0 or 1, it is a Skolem function for x_i in $F(X,Y)$.

2. $F[x_i \mapsto 1] \wedge \neg F[x_i \mapsto 0]$: For this value of variables in $X \setminus x_i \cup Y$, x_i must be 1 for F to become 1. Hence ψ must be 1 for it to become a Skolem function for x_i in $F(X,Y)$.

3. $\neg F[x_i \mapsto 1] \wedge F[x_i \mapsto 0]$: Similar to case-2, x_i must be 0 for F to become 1. Hence ψ must be 0 for it to become a Skolem function for x_i in $F(X,Y)$.

4. $F[x_i \mapsto 1] \wedge F[x_i \mapsto 0]$: Similar to case-1, irrespective of whether ψ is 0 or 1, it is a Skolem function for x_i in $F(X,Y)$.

Hence, ψ is a Skolem function for x_i in F iff $F[x_i \mapsto 1] \wedge \neg F[x_i \mapsto 0] \Rightarrow \psi$ and $\neg F[x_i \mapsto 1] \wedge F[x_i \mapsto 0] \Rightarrow \neg\psi$, i.e., $F[x_i \mapsto 1] \wedge \neg F[x_i \mapsto 0] \Rightarrow \psi$ and $\psi \Rightarrow F[x_i \mapsto 1] \vee \neg F[x_i \mapsto 0]$. □

The function $F[x_i \mapsto 0]$ (resp., $F[x_i \mapsto 1]$) is called the *negative* (resp., *positive*) *cofactor* of F with respect to x_i, and plays a central role in the study of Skolem functions for propositional formulas. In particular, it follows from Proposition 8 that $F[x_i \mapsto 1]$ is a Skolem function for x_i in F.

The above definition of a Skolem function for a single variable can be naturally extended to a vector of variables. Given $F(X,Y)$, a *Skolem function vector* for $X = (x_1, \ldots x_n)$ in F is a vector of functions $\Psi = (\psi_1, \ldots, \psi_n)$ such that $\exists x_1 \ldots \exists x_n . F \equiv (\cdots (F[x_1 \mapsto \psi_1]) \cdots [x_n \mapsto \psi_n])$. A straightforward way to obtain a Skolem function vector Ψ is to first obtain a Skolem function ψ_1 for x_1 in F, then compute $F' \equiv \exists x_1 . F$ and obtain a Skolem function ψ_2 for x_2 in F', and so on until ψ_n has been obtained. More formally, ψ_i can be computed as a Skolem function for x_i in $\exists x_1 \ldots \exists x_{i-1} . F$, starting from ψ_1 and proceeding to ψ_n. Note that $\exists x_1 \ldots \exists x_{i-1} . F$ can itself be computed as $(\cdots (F[x_1 \mapsto \psi_1]) \cdots [x_{i-1} \mapsto \psi_{i-1}])$.

Definition 2. *The "Can't-be-1" function for x_i in F, denoted $\mathtt{Cb1}[x_i](F)$, is defined to be $(\neg \exists x_1 \ldots \exists x_{i-1} . F)\,[x_i \mapsto 1]$. Similarly, the "Can't-be-0" function for x_i in F, denoted $\mathtt{Cb0}[x_i](F)$, is defined to be $(\neg \exists x_1 \ldots \exists x_{i-1} . F)\,[x_i \mapsto 0]$. When X and F are clear from the context, we use $\mathtt{Cb1}[i]$ and $\mathtt{Cb0}[i]$ for $\mathtt{Cb1}[x_i](F)$ and $\mathtt{Cb0}[x_i](F)$, respectively.*

Intuitively, in order to make F evaluate to 1, we cannot set x_i to 1 (resp. 0) whenever the valuation of $\{x_{i+1}, \ldots x_n\} \cup Y$ satisfies $\mathtt{Cb1}[i]$ (resp., $\mathtt{Cb0}[i]$). The following proposition follows from Definition 2 and from our observation about computing a Skolem function vector one component at a time.

Proposition 9. *The function vector $\Psi = (\neg \mathtt{Cb1}[1], \ldots \neg \mathtt{Cb1}[n])$ is a Skolem function vector for X in F.*

Note that the support of ψ_i in Ψ, as given by Proposition 9, is $\{x_{i+1}, \ldots x_n\} \cup Y$. If we want a Skolem function vector Ψ such that every component function has only Y (or a subset thereof) as support, this can be obtained easily by repeatedly

substituting the Skolem function for every variable x_i in all other Skolem functions where x_i appears. We often denote such a Skolem function vector as $\Psi(Y)$.

5.2 A Monolithic Composition Based Algorithm

Our algorithm is motivated in part by cofactor-based techniques for computing Skolem functions, as proposed by Jiang et al [18] and Trivedi [6]. Given $F(X,Y) = \bigwedge_{i=1}^{r} f^i(X_i,Y_i)$, the techniques of [18, 6] essentially compute a Skolem function vector $\Psi(Y)$ for X in F as shown in algorithm *MonoSkolem* (see Algorithm 11). In this algorithm, the variables in X are assumed to be ordered by their indices. While variable ordering is known to affect the difficulty of computing Skolem functions [18], we assume w.l.o.g. that the variables are indexed to represent a desirable order. We describe the variable order used in our study in Section 5.4.

MonoSkolem works in two phases. In the first phase, it implements a straightforward strategy for obtaining a Skolem function vector, as suggested by Proposition 9. Specifically, steps 3 and 4 of *MonoSkolem* build a monolithic conjunction F_i of all factors that have x_i in their support, before computing ψ_i. This restricts the scope of the quantifier for x_i to the conjunction of these factors. In Step 6, we use $\neg$`Cb1`$[i]$ for the Skolem function ψ_i. After computing ψ_i from F_i, step 7 discards the factors with x_i in their support, and introduces a single factor representing $\exists x_i.F_i$ in their place. Note that each ψ_i obtained in this manner has $\{x_{i+1},\ldots x_n\} \cup Y$ (or a subset thereof) as support. Since we want each Skolem function to have support Y, a second phase of "reverse" substitutions is needed. In this phase (see Algorithm 12), the Skolem function $\psi_n(Y)$ obtained above is substituted for x_n in $\psi_1,\ldots\psi_{n-1}$. This effectively renders all Skolem functions

independent of x_n. The process is then repeated with ψ_{n-1} substituted for x_{n-1} in $\psi_1, \ldots \psi_{n-2}$ and so on, until all Skolem functions have been made independent of $x_1, \ldots x_n$, and have only Y (or subsets thereof) as support.

MonoSkolem can be further refined by combining steps 6, 7 and by directly defining ψ_i in terms of F_i. However, we introduce the intermediate step using $\texttt{Cb0}[i]$ and $\texttt{Cb1}[i]$ to motivate their central role in our approach. Indeed instead of $\neg\texttt{Cb1}[i]$, we could use any Skolem function for x_i in F in Step 6 of the above algorithm. In fact, Jiang et al [18] compute a Skolem function for x_i in F as an interpolant of $\neg\texttt{Cb1}[i] \wedge \texttt{Cb0}[i]$ and $\texttt{Cb1}[i] \wedge \neg\texttt{Cb0}[i]$, while Trivedi [6] observes that the function $(\neg\texttt{Cb1}[i] \wedge (\texttt{Cb0}[i] \vee g)) \vee (\texttt{Cb1}[i] \wedge \texttt{Cb0}[i] \wedge h)$ serves as a Skolem function for x_i in F where h and g are arbitrary propositional functions with support in $X \setminus \{x_i\} \cup Y$. Since computing interpolants using a SAT solver is often time-intensive and does not always lead to succinct Skolem functions [18], we simply use $\neg\texttt{Cb1}[i]$ as a Skolem function in Step 6. Proposition 9 guarantees the correctness of this choice.

Let us consider an example to understand *MonoSkolem*. We will use this as a running example throughout this chapter.

Example: Let X be (x_1, x_2, x_3) and Y be (y_1, y_2). Let $F(X,Y)$ be the conjunction of factors $(y_1 \wedge x_1) \vee (\neg y_1 \wedge \neg x_1)$, $(x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2)$, $(x_2 \wedge x_3) \vee (\neg x_2 \wedge \neg x_3)$, and $(x_3 \vee y_2)$. For brevity, we will use the notation $(\alpha \Leftrightarrow \beta)$ to denote $(\alpha \wedge \beta) \vee (\neg\alpha \wedge \neg\beta)$. Thus, our factors are $(y_1 \Leftrightarrow x_1)$, $(x_1 \Leftrightarrow x_2)$, $(x_2 \Leftrightarrow x_3)$, and $(x_3 \vee y_2)$. Suppose we wish to compute Skolem function vector (ψ_1, ψ_2, ψ_3) for (x_1, x_2, x_3) in F.

In *MonoSkolem*, initially, we have $\mathsf{Factors} = \{(y_1 \Leftrightarrow x_1), (x_1 \Leftrightarrow x_2), (x_2 \Leftrightarrow x_3), (x_3 \vee y_2)\}$. Since $(y_1 \Leftrightarrow x_1)$ and $(x_1 \Leftrightarrow x_2)$ are the factors with x_1 in support, F_1

Algorithm 11: *MonoSkolem*

Input: Prop. formula $F(X,Y) = \bigwedge_{j=1}^{r} f^j(X_j,Y_j)$, where $X = (x_1,\ldots x_n)$

Output: Skolem function vector $\Psi(Y)$

`// Phase 1 of algorithm`

1 $\mathsf{Factors} := \{ f^j : 1 \leq j \leq r \}$;

2 **for** *i in* 1 *to n* **do**

3 $\quad \mathsf{FactorsWithXi} := \{ f : f \in \mathsf{Factors} \text{ and } x_i \in \mathsf{Supp}(f) \}$;

4 $\quad F_i := \bigwedge_{f \in \mathsf{FactorsWithXi}} f$;

5 $\quad \mathtt{Cb0}[i] := \neg F_i[x_i \mapsto 0]$; $\mathtt{Cb1}[i] := \neg F_i[x_i \mapsto 1]$;

6 $\quad \psi_i := \neg \mathtt{Cb1}[i]$;

$\quad$ `// In general,` ψ_i `is an interpolant of` $\neg \mathtt{Cb1}[i] \wedge \mathtt{Cb0}[i]$

$\quad$ `// and` $\mathtt{Cb1}[i] \wedge \neg \mathtt{Cb0}[i]$

7 $\quad \mathsf{Factors} := (\mathsf{Factors} \setminus \mathsf{FactorsWithXi}) \cup \{F_i[x_i \mapsto \psi_i]\}$;

`// Phase 2 of algorithm`

8 **return** *ReverseSubstitute*$(\psi_1,\ldots,\psi_n)$;

Algorithm 12: *ReverseSubstitute*

Input: Functions $\psi_1(x_2,\ldots,x_n,Y), \psi_2(x_3,\ldots,x_n,Y),\ldots,\psi_n(Y)$

Output: Function vector $\Psi(Y)$

1 **for** $i = n$ *downto* 2 **do**

2 $\quad$ **for** $j = i-1$ *downto* 1 **do** $\psi_j = \psi_j[x_i \mapsto \psi_i]$;

3 **return** $\Psi(Y) = (\psi_1(Y),\ldots,\psi_n(Y))$;

is $(y_1 \Leftrightarrow x_1) \wedge (x_1 \Leftrightarrow x_2)$. Note that $F_1[x_1 \mapsto 0]$ is $\neg y_1 \wedge \neg x_2$ and $F_1[x_1 \mapsto 1]$ is $y_1 \wedge x_2$. Hence, $\mathtt{Cb0}[1]$ is $y_1 \vee x_2$, $\mathtt{Cb1}[1]$ is $\neg y_1 \vee \neg x_2$, and ψ_1 is $y_1 \wedge x_2$. Replacing

the factors $(y_1 \Leftrightarrow x_1)$ and $(x_1 \Leftrightarrow x_2)$ by the single factor $F_1[x_1 \mapsto \psi_1]$, we have $\mathsf{Factors} = \{(y_1 \Leftrightarrow y_1 \wedge x_2) \wedge (y_1 \wedge x_2 \Leftrightarrow x_2), (x_2 \Leftrightarrow x_3), (x_3 \vee y_2)\}$.

Next, Skolem function ψ_2 is computed. Now F_2 is $(y_1 \Leftrightarrow y_1 \wedge x_2) \wedge (y_1 \wedge x_2 \Leftrightarrow x_2) \wedge (x_2 \Leftrightarrow x_3)$. Note that $F_2[x_2 \mapsto 0]$ is $\neg y_1 \wedge \neg x_3$ and $F_2[x_2 \mapsto 1]$ is $y_1 \wedge x_3$. Hence, $\mathtt{Cb0}[2]$ is $y_1 \vee x_3$, $\mathtt{Cb1}[2]$ is $\neg y_1 \vee \neg x_3$. ψ_2 is $y_1 \wedge x_3$, and $\mathsf{Factors}$ gets changed to $\{(y_1 \Leftrightarrow y_1 \wedge x_3) \wedge (y_1 \wedge x_3 \Leftrightarrow x_3), (x_3 \vee y_2)\}$. Finally, F_3 is $(y_1 \Leftrightarrow y_1 \wedge x_3) \wedge (y_1 \wedge x_3 \Leftrightarrow x_3) \wedge (x_3 \vee y_2)$. $F_3[x_3 \mapsto 0]$ is $\neg y_1 \wedge y_2$ and $F_3[x_3 \mapsto 1]$ is y_1. Also $\mathtt{Cb0}[3]$ is $y_1 \vee \neg y_2$, $\mathtt{Cb1}[3]$ is $\neg y_1$, and ψ_3 is y_1. $\mathsf{Factors}$ becomes $\{(y_1 \vee y_2)\}$.

This completes the first phase of *MonoSkolem*. Thus, after the first phase, we have $(\psi_1, \psi_2, \psi_3) = (y_1 \wedge x_2,\ y_1 \wedge x_3,\ y_1)$. In the second phase, first ψ_3 is substituted for x_3 in ψ_2. This makes $\psi_2 = y_1$. Now ψ_2 is substituted for x_2 in ψ_1, which makes $\psi_1 = y_1$. Hence, after the second phase, we have $(\psi_1, \psi_2, \psi_3) = (y_1, y_1, y_1)$.

Observe that *MonoSkolem* works with a *monolithic* conjunction (F_i) of factors that have x_i in their support. Specifically, it composes each such monolithic conjunction F_i with a cofactor of F_i in Step 7 to eliminate quantifiers sequentially. This can lead to large memory footprints and more time-outs when used with medium to large benchmarks as confirmed by our experiments. This motivates us to ask if we can develop a cofactor-based algorithm that does not suffer from the above drawbacks of *MonoSkolem*.

5.3 CEGAR for Generating Skolem Functions

We now present a new CEGAR [105] algorithm for generating Skolem function vectors, that exploits the factored form of $F(X,Y)$. Like *MonoSkolem*, our new

algorithm, named *CegarSkolem*, works in two phases, and assumes that the variables in X are ordered by their indices. The first phase of the algorithm consists of the core abstraction-refinement part, and computes a Skolem function vector $(\psi_1, \ldots \psi_n)$, where ψ_i has $\{x_{i+1}, \ldots x_n\} \cup Y$, or a subset thereof, as support. Unlike in *MonoSkolem*, this phase avoids composing monolithic conjunctions of factors, yielding simpler Skolem functions. The second phase of the algorithm performs reverse substitutions, similar to that in *MonoSkolem*.

Before describing the details of *CegarSkolem*, we introduce some additional notation and terminology. Given propositional functions f and g, we say that f *refines* g and g *abstracts* f iff $f \Rightarrow g$. Given $F(X,Y)$ and a vector of functions $\Psi^A = (\psi_1^A, \ldots \psi_n^A)$, we say that Ψ^A is an *abstract Skolem function vector* for X in F iff there exists a Skolem function vector $\Psi = (\psi_1, \ldots \psi_n)$ for X in F such that ψ_i^A abstracts ψ_i, for every $i \in \{1, \ldots n\}$. Instead of using $\texttt{Cb0}[i]$ and $\texttt{Cb1}[i]$ to compute Skolem functions, as was done in *MonoSkolem*, we now use their *refinements*, denoted $\texttt{r0}[i]$ and $\texttt{r1}[i]$ respectively, to compute abstract Skolem functions. For convenience, we represent $\texttt{r0}[i]$ and $\texttt{r1}[i]$ as sets of implicitly disjoined functions. Thus, if $\texttt{r1}[i]$, viewed as a set, is $\{g_1, g_2\}$, then it is $g_1 \vee g_2$ when viewed as a function. We abuse notation and use $\texttt{r1}[i]$ (respectively, $\texttt{r0}[i]$) to denote a set of functions or their disjunction, depending on the context.

5.3.1 Overview of Our CEGAR Algorithm

Algorithm *CegarSkolem* has two phases. The first phase consists of a CEGAR loop, while the second phase does reverse substitutions. The CEGAR loop has the following steps.

– **Initial abstraction and refinement.** This step involves constructing refinements of $\texttt{Cb0}[i]$ and $\texttt{Cb1}[i]$ for every x_i in X. By Proposition 9, this gives an initial abstract Skolem function vector Ψ^A. This step is implemented in Algorithm 13 (*InitAbsRef*), which processes individual factors of $F(X,Y) = \bigwedge_{j=1}^{r} f^j(X_j,Y_j)$ separately, without considering their conjunction. As a result, this step is time and memory efficient if the individual factors are small and simple.

– **Termination Condition.** Once *InitAbsRef* has computed Ψ^A, we check whether Ψ^A is already a Skolem function vector. This is achieved by constructing an appropriate propositional formula ε, called the "error formula" for Ψ^A (details in Subsection 5.3.3), and checking for its satisfiability. An unsatisfiable formula implies that Ψ^A is a Skolem function vector, and we are done with the first phase. Otherwise, a satisfying assignment π of ε is used to improve the current refinements of $\texttt{Cb1}[i]$ and $\texttt{Cb0}[i]$ for suitable variables x_i in X.

– **Counterexample guided abstraction and refinement.** This step is implemented in Algorithm 14:*UpdateAbsRef,* and leads to a refinement of the abstract Skolem function vector Ψ^A.

Thus, the overall CEGAR loop starts with the first step and then repeats the second and third steps until a Skolem function vector is obtained. In the next three subsections, we discuss the algorithms implementing these steps in detail.

Algorithm 13: *InitAbsRef*

Input: Prop. formula $F(X,Y) = \bigwedge_{j=1}^{r} f^j(X_j,Y_j)$, where $X = (x_1,\ldots x_n)$

Output: Abstract Skolem function vector $\Psi^{\mathsf{A}} = (\psi_1^A,\ldots,\psi_n^A)$, and refinements $\mathtt{r0}[i]$ and $\mathtt{r1}[i]$ for each x_i in X

1 **for** *i in* 1 *to n* **do**
2 $\mathtt{r0}[i] := \emptyset$; $\mathtt{r1}[i] := \emptyset$; `// Initializing`
3 **for** *j in* 1 *to r* **do**
4 $f := f^j$; `// Consider factor separately`
5 **for** *i in* 1 *to n* **do**
6 **if** $x_i \in \mathsf{Supp}(f)$ **then**
7 $\mathtt{r0}[i] := \mathtt{r0}[i] \cup \{\neg f[x_i \mapsto 0]\}$;
8 $\mathtt{r1}[i] := \mathtt{r1}[i] \cup \{\neg f[x_i \mapsto 1]\}$;
 `// Skolem function for` x_i `in` f
9 $\psi_{i,f} := f[x_i \mapsto 1]$;
10 $f := f[x_i \mapsto \psi_{i,f}]$; `//` $f[x_i \mapsto \psi_{i,f}] \equiv \exists x_i.\, f$
11 **for** *i in* 1 *to n* **do**
12 $\psi_i^A := \neg\mathtt{r1}[i]$;
 `// Interpreting` $\mathtt{r1}[i]$ `as a function`
13 **return** $\Psi^{\mathsf{A}}=(\psi_1^A,\ldots\psi_n^A)$ *and* $\mathtt{r0}[i],\mathtt{r1}[i]$ *for each* $x_i \in X$

5.3.2 Initial Abstraction and Refinement

Algorithm *InitAbsRef* (see Algorithm 13) starts by initializing each $\texttt{r1}[i]$ and $\texttt{r0}[i]$, viewed as sets, to the empty set. Subsequently, it considers each factor f in $\bigwedge_{j=1}^{r} f^j(X_j, Y_j)$, and determines the contribution of f to $\texttt{Cb0}[i]$ and $\texttt{Cb1}[i]$, for every x_i in the support of f. Specifically, if $x_i \in \mathsf{Supp}(f)$, the contribution of f to $\texttt{Cb0}[i]$ is $(\neg\exists x_1 \ldots \exists x_{i-1}.\, f)\,[x_i \mapsto 0]$, and its contribution to $\texttt{Cb1}[i]$ is $(\neg\exists x_1 \ldots \exists x_{i-1}.\, f)\,[x_i \mapsto 1]$. These contributions are accumulated in the sets $\texttt{r0}[i]$ and $\texttt{r1}[i]$, respectively, and then x_i is existentially quantified from f and the process repeated with the next variable in the support of f. Once the contributions from all factors are accumulated in $\texttt{r0}[i]$ and $\texttt{r1}[i]$ for each x_i in X, *InitAbsRef* computes an abstract Skolem function ψ_i^A for each x_i in F by complementing $\texttt{r1}[i]$, interpreted as a disjunction of functions.

Example: Consider the execution of *InitAbsRef* in our example of computing Skolem function vector (ψ_1, ψ_2, ψ_3) for (x_1, x_2, x_3) in $(y_1 \Leftrightarrow x_1) \wedge (x_1 \Leftrightarrow x_2) \wedge (x_2 \Leftrightarrow x_3) \wedge (x_3 \vee y_2)$. Initially $\texttt{r0}[i] = \emptyset$ and $\texttt{r1}[i] = \emptyset$, for $1 \leq i \leq 3$. *InitAbsRef* now considers the first factor $(y_1 \Leftrightarrow x_1)$, and determines the contribution of this factor to $\texttt{Cb0}[1]$ and $\texttt{Cb1}[1]$. Since $(y_1 \Leftrightarrow x_1)[x_1 \mapsto 0]$ is $\neg y_1$ and $(y_1 \Leftrightarrow x_1)[x_1 \mapsto 1]$ is y_1, the contribution of $(y_1 \Leftrightarrow x_1)$ to $\texttt{Cb0}[1]$ is y_1 and the contribution to $\texttt{Cb1}[1]$ is $\neg y_1$. Hence $\texttt{r0}[1]$ gets changed to $\{y_1\}$ and $\texttt{r1}[1]$ gets changed to $\{\neg y_1\}$. The factor $(y_1 \Leftrightarrow x_1)$ now gets changed to $\exists x_1.\,(y_1 \Leftrightarrow x_1)$, i.e., 1.

InitAbsRef then proceeds with the next factor $(x_1 \Leftrightarrow x_2)$, and determines its contribution to $\texttt{Cb0}[1]$ and $\texttt{Cb1}[1]$. Note that the contribution of $(x_1 \Leftrightarrow x_2)$ to $\texttt{Cb0}[1]$ is x_2 and contribution to $\texttt{Cb1}[1]$ is $\neg x_2$, which are accumulated in $\texttt{r0}[1]$ and $\texttt{r1}[1]$ respectively. Thus $\texttt{r0}[1]$ becomes $\{y_1, x_2\}$ and $\texttt{r1}[1]$ becomes $\{\neg y_1, \neg x_2\}$. The factor $(x_1 \Leftrightarrow x_2)$ gets changed to $\exists x_1.\,(x_1 \Leftrightarrow x_2)$, i.e., 1. Subsequently, the

factor $(x_2 \Leftrightarrow x_3)$ is considered. $\texttt{r0}[2]$ and $\texttt{r1}[2]$ are updated to $\{x_3\}$ and $\{\neg x_3\}$ respectively. Finally, the factor $(x_3 \vee y_2)$ is considered, and $\texttt{r0}[3]$, $\texttt{r1}[3]$ are updated to $\{\neg y_2\}$ and $\{0\}$.

Thus, finally we have, $\texttt{r0}[1] = \{y_1, x_2\}$, $\texttt{r1}[1] = \{\neg y_1, \neg x_2\}$, $\texttt{r0}[2] = \{x_3\}$, $\texttt{r1}[2] = \{\neg x_3\}$, $\texttt{r0}[3] = \{\neg y_2\}$, and $\texttt{r1}[3] = \{0\}$, when interpreted as sets. When interpreted as disjunctions, $\texttt{r0}[1] = y_1 \vee x_2$, $\texttt{r1}[1] = \neg y_1 \vee \neg x_2$, $\texttt{r0}[2] = x_3$, $\texttt{r1}[2] = \neg x_3$, $\texttt{r0}[3] = \neg y_2$, and $\texttt{r1}[3] = 0$. Since the abstract Skolem function ψ_i^A for each x_i in F is obtained by complementing $\texttt{r1}[i]$, we have, $\psi_1^A = y_1 \wedge x_2$, $\psi_2^A = x_3$, and $\psi_3^A = 1$.

Recall from Section 5.2 that, for this example, $\texttt{Cb0}[1] = y_1 \vee x_2$, $\texttt{Cb1}[1] = \neg y_1 \vee \neg x_2$, $\texttt{Cb0}[2] = y_1 \vee x_3$, $\texttt{Cb1}[2] = \neg y_1 \vee \neg x_3$, $\texttt{Cb0}[3] = y_1 \vee \neg y_2$, and $\texttt{Cb1}[3] = \neg y_1$. Observe that $\texttt{r0}[i]$ and $\texttt{r1}[i]$ are refinements of $\texttt{Cb0}[i]$ and $\texttt{Cb1}[i]$ respectively, for each x_i, since $\texttt{r0}[i] \Rightarrow \texttt{Cb0}[i]$ and $\texttt{r1}[i] \Rightarrow \texttt{Cb1}[i]$, when $\texttt{r0}[i]$ and $\texttt{r1}[i]$ are interpreted as disjunctions.

Note that executing steps 4 through 10 of *InitAbsRef* for a specific factor f is operationally similar to executing steps 1 through 7 of *MonoSkolem* with a singleton set of factors, i.e. $\mathsf{Factors} = \{f\}$. This highlights the key difference between *InitAbsRef* and *MonoSkolem*: while *MonoSkolem* works with monolithic conjunctions of factors and their compositions, *InitAbsRef* works with individual factors, without ever considering their conjunctions. Lemma 8 asserts the correctness of *InitAbsRef*.

Lemma 8. *The vector* Ψ^{A} *computed by* InitAbsRef *is an abstract Skolem function vector for* X *in* $F(X,Y)$. *In addition,* $\texttt{r0}[i]$ *and* $\texttt{r1}[i]$ *computed by* InitAbsRef *are refinements of* $\texttt{Cb0}[i](F)$ *and* $\texttt{Cb1}[i](F)$ *for every* x_i *in* X.

Proof of Lemma 8. Consider the ordered pair (j,i) of loop indices correspond-

ing to the nested loops in steps $3-10$ and $5-10$ of algorithm *InitAbsRef*. Every update of $\texttt{r0}[i]$ and $\texttt{r1}[i]$ in steps 7 and 8 of *InitAbsRef* can be associated with a unique ordered pair of loop indices. Define a linear ordering $\preceq$ on the loop index pairs as: $(j,i) \preceq (j',i')$ iff $j < j'$, or $j = j'$ and $i \leq i'$. Note that this represents the ordering of loop index pairs in successive iterations of the loop in steps $5-10$ of *InitAbsRef*. We use induction on (j,i), ordered by $\preceq$, to show that $\texttt{r0}[i]$ and $\texttt{r1}[i]$, as computed by *InitAbsRef*, are refinements of $\texttt{Cb0}[i]$ and $\texttt{Cb1}[i]$. The base case follows from the initialization in steps 1 and 2 of *InitAbsRef*. To prove the inductive step, consider an update of $\texttt{r0}[i]$ and $\texttt{r1}[i]$ in steps 7 and 8, respectively, of *InitAbsRef*. The function f used in steps 7 and 8 is easily seen to be $\exists x_1 \ldots \exists x_{i-1}.\, f^j$. Since f^j is a factor of F, we also have $F \Rightarrow f^j$. It follows that $\exists x_1 \ldots \exists x_{i-1}.F \;\Rightarrow\; \exists x_1 \ldots \exists x_{i-1}.\, f^j \;\equiv\; f$. Taking the contrapositive gives $\neg f \;\Rightarrow\; \neg\exists x_1 \ldots \exists x_{i-1}.F$. Therefore, $\neg f[x_i \mapsto a] \Rightarrow (\neg\exists x_1 \ldots \exists x_{i-1}.F)[x_i \mapsto a]$ for every propositional constant a. Recalling the definitions of $\texttt{Cb0}[i]$ and $\texttt{Cb1}[i]$, we get $\neg f[x_i \mapsto 0] \Rightarrow \texttt{Cb0}[i]$ and $\neg f[x_i \mapsto 1] \Rightarrow \texttt{Cb1}[1]$. By the inductive hypothesis, $\texttt{r0}[i]$ and $\texttt{r1}[i]$ are refinements of $\texttt{Cb0}[i]$ and $\texttt{Cb1}[i]$ prior to executing step 7 of *InitAbsRef*. Therefore, the updated values of $\texttt{r0}[i]$ and $\texttt{r1}[i]$, as computed in steps 7 and 8 of *InitAbsRef*, are also refinements of $\texttt{Cb0}[i]$ and $\texttt{Cb1}[i]$. This completes the induction.

Since $\texttt{r1}[i] \Rightarrow \texttt{Cb1}[i]$ for every x_i in X when we reach step 11 of *InitAbsRef*, it follows from Proposition 9 that $\psi_i^A = \neg\texttt{r1}[i]$ abstracts a Skolem function for x_i in F. Hence, Ψ^A, as computed by *InitAbsRef*, is an abstract Skolem function vector for X in F. $\square$

5.3.3 Termination Condition

Given $F(X,Y)$ and an abstract Skolem function vector Ψ^A, it may happen that Ψ^A is already a Skolem function vector for X in F. We therefore check if Ψ^A is a Skolem function vector before refinement. Towards this end, we define the *error formula* for Ψ^A as $F(X',Y) \wedge \bigwedge_{i=1}^{n}(x_i \Leftrightarrow \psi_i^A) \wedge \neg F(X,Y)$, where $X'=(x'_1,\ldots x'_n)$ is a sequence of fresh variables with no variable in common with X. The idea is that the first term in the error formula checks the satisfiability of $\exists X.\exists Y.F(X,Y)$. If it is indeed satisfiable, then the second term assigns the values of the variables to the values given by the abstract Skolem functions and then the third term checks if this assignment violates the validity of the formula. Thus,

Lemma 9. *The error formula for Ψ^A is unsatisfiable iff Ψ^A is a Skolem function vector for X in F.*

Example of Lemma 9: In our example, the error formula is $(y_1 \Leftrightarrow x'_1) \wedge (x'_1 \Leftrightarrow x'_2) \wedge (x'_2 \Leftrightarrow x'_3) \wedge (x'_3 \vee y_2) \wedge (x_1 \Leftrightarrow y_1 \wedge x_2) \wedge (x_2 \Leftrightarrow x_3) \wedge (x_3 \Leftrightarrow 1) \wedge \neg((y_1 \Leftrightarrow x_1) \wedge (x_1 \Leftrightarrow x_2) \wedge (x_2 \Leftrightarrow x_3) \wedge (x_3 \vee y_2))$. Note that the error formula is satisfiable. A satisfying assignment is $(y_1 = 0)$, $(y_2 = 1)$, $(x'_1 = 0)$, $(x'_2 = 0)$, $(x'_3 = 0)$, $(x_1 = 0)$, $(x_2 = 1)$, $(x_3 = 1)$. It can be observed that $(y_1 \wedge x_2, x_3, 1)$ is not a Skolem function vector for X in F.

Proof of Lemma 9. Let ε be the error formula for Ψ^A. Suppose ε is unsatisfiable. By definition of ε, $F(X',Y) \wedge \bigwedge_{i=1}^{n}(x_i \Leftrightarrow \psi_i^A) \wedge \neg F(X,Y)$, where $X'=(x'_1,\ldots x'_n)$ is unsatisfiable. This implies that $F(X',Y) \wedge \neg F'(Y)$ is unsatisfiable, where $F'(Y)$ denotes $\big(\cdots\big(F[x_1 \mapsto \psi_1^A]\big)\cdots[x_n \mapsto \psi_n^A]\big)$. Thus $\exists Y.\exists X'.\,(F(X',Y) \wedge \neg F'(Y))$ is false, i.e., $\forall Y.\forall X'.\,(F(X',Y) \Rightarrow F'(Y))$ is true. This implies that $\forall Y.\exists X'.(F(X',Y)$

$\Rightarrow F'(Y))$ is true, i.e., $\forall Y.(\exists X'.F(X',Y) \Rightarrow F'(Y))$ is true. Therefore, Ψ^A is a Skolem function vector for X in F.

Suppose π is a satisfying assignment of ε. By definition of ε, π is a satisfying assignment of $F(X',Y)$ and of $\bigwedge_{i=1}^{n}\left(x_i \Leftrightarrow \psi_i^A\right) \wedge \neg F(X,Y)$, considered separately. Thus, the values of $x_1,\ldots x_n$ given by $\psi_1^A,\ldots \psi_n^A$ respectively, cause F to evaluate to 0 for the valuation of Y in π. However, there exists a valuation of X (viz. same as that of X' in π) that causes F to evaluate to 1 for the same valuation of Y in π. Hence, Ψ^A is not a Skolem function vector for X in F, as witnessed by the valuation of Y in π. □

Satisfiability of the error formula can be checked using any SAT solver. If the error formula turns out to be satisfiable, we use a satisfying assignment of the formula to refine Ψ^A, as explained below.

5.3.4 CounterExample-Guided Abstraction and Refinement

Let ε be the error formula for Ψ^A, and let π be a satisfying assignment of ε. We call π a *counterexample* of the claim that Ψ^A is a Skolem function vector. For every variable $v \in X' \cup X \cup Y$, we use $\pi(v)$ to denote the value of v in π. Satisfiability of ε implies that we need to refine at least one abstract Skolem function ψ_i^A in Ψ^A to make it a Skolem function vector. Since ψ_i^A is $\neg\mathtt{r1}[i]$ in our approach, refining ψ_i^A can be achieved by computing an improved (i.e. more abstract) version of $\mathtt{r1}[i]$. Algorithm *UpdateAbsRef* implements this idea by using π to determine which $\mathtt{r1}[i]$ should be rendered abstract by adding appropriate functions to $\mathtt{r1}[i]$, viewed as a set.

Before delving into the details of *UpdateAbsRef*, we state some key results.

In the following, we use $\pi \models f$ to denote that the formula f evaluates to 1 when the variables in $\mathsf{Supp}(f)$ are set to values given by π. If $\pi \models f$, we also say f evaluates to 1 under π. We use $\mathtt{r0}[i]_{init}$ and $\mathtt{r1}[i]_{init}$ to refer to $\mathtt{r0}[i]$ and $\mathtt{r1}[i]$, as computed by algorithm *InitAbsRef*. Since *UpdateAbsRef* only adds to $\mathtt{r1}[i]$ and $\mathtt{r0}[i]$, viewed as sets, it is easy to see that $\mathtt{r0}[i]_{init} \Rightarrow \mathtt{r0}[i]$ and $\mathtt{r1}[i]_{init} \Rightarrow \mathtt{r1}[i]$, viewed as functions (recall these functions are simply disjunctions of elements in the corresponding sets).

The following Lemma forms the basis for the refinement that we perform in the algorithm *UpdateAbsRef*.

Lemma 10. *Let π be a satisfying assignment of the error formula ε for Ψ^{A}. Then the following hold.*

(a) $\pi \models \neg\mathtt{Cb0}[n] \vee \neg\mathtt{Cb1}[n]$.

(b) There exists $k \in \{1,\ldots n-1\}$ such that $\pi \models \mathtt{r1}[k] \wedge \mathtt{r0}[k]$.

(c) There exists no Skolem function vector $\Psi = (\psi_1,\ldots \psi_n)$ such that $\psi_j \Leftrightarrow \psi_j^A$ for all j in $\{k+1,\ldots n\}$.

(d) There exists $l \in \{k+1,\ldots n\}$ such that $x_l = 1$ in π, and $\pi \models \mathtt{Cb1}[l] \wedge \neg\mathtt{r0}[l]$.

Example of Lemma 10: Recall that our running example, π is $(y_1 = 0)$, $(y_2 = 1)$, $(x'_1 = 0)$, $(x'_2 = 0)$, $(x'_3 = 0)$, $(x_1 = 0)$, $(x_2 = 1)$, $(x_3 = 1)$. Since $\mathtt{Cb0}[3] = y_1 \vee \neg y_2$ and $\mathtt{Cb1}[3] = \neg y_1$, clearly $\pi \models \neg\mathtt{Cb0}[3] \vee \neg\mathtt{Cb1}[3]$. Recall that $\mathtt{r0}[1] = y_1 \vee x_2$ and $\mathtt{r1}[1] = \neg y_1 \vee \neg x_2$. Hence, $\pi \models \mathtt{r1}[1] \wedge \mathtt{r0}[1]$ and $k = 1$. Note that we have $(\psi_1^A, \psi_2^A, \psi_3^A) = (y_1 \wedge x_2,\ x_3,\ 1)$. It can be observed that there exists no Skolem function vector $\Psi = (\psi_1, \psi_2, \psi_3)$ such that $\psi_2 \Leftrightarrow \psi_2^A$ and $\psi_3 \Leftrightarrow \psi_3^A$.

Finally observe that since $\mathtt{r0}[3]$ is $\neg y_2$, $\mathtt{Cb1}[3] \wedge \neg \mathtt{r0}[3]$ is $\neg y_1 \wedge y_2$. Also $\pi \models \mathtt{Cb1}[3] \wedge \neg \mathtt{r0}[3]$ and $l = 3$.

For clarity of exposition, we postpone the proof of Lemma 10 to the end of this section.

Algorithm 14 (*UpdateAbsRef*) uses Lemma 10 to compute abstract versions of $\mathtt{r0}[i]$ and $\mathtt{r1}[i]$, and a refined version of Ψ^{A}, when Ψ^{A} is not a Skolem function vector. The algorithm takes as inputs the current versions of $\mathtt{r0}[i]$ and $\mathtt{r1}[i]$ for all x_i in X, and a satisfying assignment π of the error formula for the current version of Ψ^{A}. Since $\pi \models F(X',Y)$ and $\pi \models \neg F(X,Y)$, and since the value of every x_i in π is given by ψ_i^A, there exists at least one ψ_l^A, where $l \in \{1,\ldots n\}$, that fails to generate the right value of x_l when the value of Y is as given by π. *UpdateAbsRef* works by identifying such an index l and refining ψ_l^A. Since $\psi_i^A = \neg \mathtt{r1}[i]$, the refinement of ψ_l^A is effected by updating (abstracting) the corresponding $\mathtt{r1}[l]$ set. In fact, the algorithm may, in general, end up abstracting not only $\mathtt{r1}[l]$, but several $\mathtt{r0}[i]$ and $\mathtt{r1}[i]$ as well in a sound manner.

As shown in Algorithm 14, *UpdateAbsRef* first finds the largest index k such that $\pi \models \mathtt{r0}[k] \wedge \mathtt{r1}[k]$. Lemma 10b guarantees the existence of such an index in $\{1,\ldots n-1\}$. We assume access to a function called *Generalize* that takes as arguments an assignment π and a function φ such that $\pi \models \varphi$, and returns a function ξ that generalizes π while satisfying φ. More formally, if $\xi = \mathit{Generalize}(\pi, \varphi)$, then $\mathsf{Supp}(\xi) \subseteq \mathsf{Supp}(\varphi)$, $\pi \models \xi$ and $\xi \Rightarrow \varphi$. Thus, in steps 2 and 3 of *UpdateAbsRef*, we compute generalizations of π that satisfy $\mathtt{r0}[k]$ and $\mathtt{r1}[k]$, respectively. The function μ computed in step 4 is therefore such that $\pi \models \mu$ and $\mu \Rightarrow \mathtt{r0}[k] \wedge \mathtt{r1}[k]$.

The function *Generalize*(π, φ) can be implemented in several ways. Since $\pi \models \varphi$, we could return a conjunction of literals corresponding to the assignment

Algorithm 14: *UpdateAbsRef*

Input: $\mathtt{r0}[i]$ and $\mathtt{r1}[i]$ for all x_i in X,

Satisfying assignment π of error formula, i.e.

$F(X',Y) \wedge \bigwedge_{i=1}^{n} \left(x_i \Leftrightarrow \psi_i^A\right) \wedge \neg F(X,Y)$

Output: Improved (i.e. refined) $\Psi^{\mathsf{A}} = (\psi_1^A, \ldots \psi_n^A)$, and

Improved (i.e. abstracted) $\mathtt{r0}[i]$ and $\mathtt{r1}[i]$ for all x_i in X

```
1  k := largest j such that π satisfies r0[j] ∧ r1[j];
2  μ0 := Generalize(π, r0[k]);
3  μ1 := Generalize(π, r1[k]);
4  μ := μ0 ∧ μ1;
5  l := k + 1;
6  while true do                    // current guess: ψ_l^A to be refined
7      if x_l ∈ Supp(μ) then
8          if x_l = 1 in π then
9              μ1 := μ[x_l ↦ 1];
10             r1[l] := r1[l] ∪ {μ1};
11             if π satisfies r0[l] then
12                 μ0 := Generalize(π, r0[l]);
13                 μ := μ0 ∧ μ1;
14             else
15                 break;
16         else
17             μ0 := μ[x_l ↦ 0];
18             r0[l] := r0[l] ∪ {μ0};
19             μ1 := Generalize(π, r1[l]);
20             μ := μ0 ∧ μ1;
21     l := l + 1 ;
22 Ψ^A = (¬r1[1], ... ¬r1[n]);
23 return r0[i] and r1[i] for all x_i in X, and Ψ^A
```

π, or the function φ itself. From our experiments, it appears that the first option leads to low memory requirements, but increased run-time due to large number of invocations of *UpdateAbsRef*. The other option leads to higher memory requirements, but reduced run-time due to fewer invocations of *UpdateAbsRef*. For our study, we let *Generalize*(π, $\mathtt{r1}[k]$) return one of the functions in $\mathtt{r1}[k]$ (viewed as a set) that evaluate to 1 under π. We follow a similar strategy for *Generalize*(π, $\mathtt{r0}[k]$) as well. This appears to give us a reasonable tradeoff between time and space requirements.

Example (Continued): In the example, $k = 1$ since (i) $\pi \models \mathtt{r1}[1] \wedge \mathtt{r0}[1]$, (ii) $\pi \not\models \mathtt{r1}[2] \wedge \mathtt{r0}[2]$, and (iii) $\pi \not\models \mathtt{r1}[3] \wedge \mathtt{r0}[3]$. Since $\mathtt{r1}[1] = \neg y_1 \vee \neg x_2$, $\mathtt{r0}[1] = y_1 \vee x_2$, $\pi \models \neg y_1$ and $\pi \models x_2$, *Generalize*(π, $\mathtt{r1}[1]$) returns $\neg y_1$, and *Generalize*(π, $\mathtt{r0}[1]$) returns x_2. Hence μ at step 4 is $x_2 \wedge \neg y_1$. Note that $\pi \models \mu$ and $\mu \Rightarrow \mathtt{r0}[1] \wedge \mathtt{r1}[1]$.

Using Definition 2, $\mathtt{Cb1}[k] \wedge \mathtt{Cb0}[k]$ is equivalent to $(\neg\exists x_1. \ldots \exists x_{k-1}. F)\,[x_k \mapsto 0] \wedge (\neg\exists x_1. \ldots \exists x_{k-1}. F)\,[x_k \mapsto 1]$, which is equivalent to $\neg\exists x_1. \ldots \exists x_k. F$. Since $\mathtt{r1}[k] \Rightarrow \mathtt{Cb1}[k]$ and $\mathtt{r0}[k] \Rightarrow \mathtt{Cb0}[k]$, this implies that $\mathtt{r0}[k] \wedge \mathtt{r1}[k] \Rightarrow \neg\exists x_1. \ldots \exists x_k. F$. Since $\mu \Rightarrow \mathtt{r0}[k] \wedge \mathtt{r1}[k]$, we have, $\mu \Rightarrow \neg\exists x_1. \ldots \exists x_k. F$. This means that any abstract Skolem function vector that produces values of $x_1, \ldots x_n$ (given value of Y as in π) for which μ evaluates to 1, cannot be a Skolem function vector. Since the support of μ is $\{x_{k+1}, \ldots x_n\} \cup Y$, one of the abstract Skolem functions $\psi^A_{k+1}, \ldots \psi^A_n$ must be refined.

The loop in steps 6–21 of *UpdateAbsRef* tries to identify an abstract Skolem function ψ^A_l to be refined, by iterating l from $k+1$ to n. Clearly, if $x_l \notin \mathsf{Supp}(\mu)$, the value of ψ^A_l under π is of no consequence in evaluating μ, and we ignore such variables. If $x_l \in \mathsf{Supp}(\mu)$ and if $x_l = 1$ in π, then $\pi \models \mu[x_l \mapsto 1]$. Moreover, it can be observed that μ at this step (i.e. at step 9) is such that $\mu \Rightarrow$

$\neg\exists x_1.\ldots\exists x_{l-1}.F$. Hence, $\mu[x_l \mapsto 1] \Rightarrow (\neg\exists x_1.\ldots\exists x_{l-1}.F)[x_l \mapsto 1]$. Recalling the definition of $\texttt{Cb1}[l]$, we have $\mu[x_l \mapsto 1] \Rightarrow \texttt{Cb1}[l]$, and therefore $\mu[x_l \mapsto 1]$ can be added to $\texttt{r1}[l]$ (viewed as a set) yielding a more abstract version of $\texttt{r1}[l]$. Steps 8–10 of *UpdateAbsRef* implement this update of $\texttt{r1}[l]$.

Example (Continued): Recall that $k = 1$ and μ at step 4 is $x_2 \wedge \neg y_1$. $l = 2$, and since $x_2 \in \mathsf{Supp}(\mu)$ and $x_2 = 1$ in π, we reach step 9. Note that $\mu[x_2 \mapsto 1] = \neg y_1$ $\Rightarrow \texttt{Cb1}[2] = (\neg y_1 \vee \neg x_3)$. After adding $\neg y_1$, $\texttt{r1}[2]$ becomes $\{\neg x_3, \neg y_1\}$ viewed as a set, and $\neg x_3 \vee \neg y_1$ when viewed as a disjunction.

Since $\pi \models \mu[x_l \mapsto 1]$, we have $\pi \models \texttt{r1}[l]$ after step 10. If it so happens that $\pi \models \texttt{r0}[l]$ as well, then we have $\pi \models \texttt{r0}[l] \wedge \texttt{r1}[l]$, where $\texttt{r1}[l]$ refers to the updated refinement of $\texttt{Cb1}[l]$. In this case, we have effectively found an index $l > k$ such that $\pi \models \texttt{r0}[k] \wedge \texttt{r1}[k]$. We can therefore repeat our algorithm starting with l instead of k. Steps 11–13 followed by step 21 of algorithm *UpdateAbsRef* effectively implement this. Steps 11–13 update μ, and step 21 increments l by 1, so that the implication $\mu \Rightarrow \neg\exists x_1.\ldots\exists x_{l-1}.F$ is preserved. If, on the other hand, $\pi \not\models \texttt{r0}[k]$, then we have found an l that satisfies the conditions in Lemma 10d. We exit the search for an abstract Skolem function in this case (see steps 14–15).

Example (Continued): As we saw, at step 10, $\texttt{r1}[2]$ is updated to $\neg x_3 \vee \neg y_1$. Note that $\texttt{r0}[2] = x_3$, and $\pi \models x_3$. Thus we reach step 12. *Generalize*(π, $\texttt{r0}[2]$) returns x_3. Hence μ at step 13 is $x_3 \wedge \neg y_1$. The value of l is now incremented to 3, and the next iteration of the loop starts. Since $x_3 \in \mathsf{Supp}(\mu)$ and $x_3 = 1$ in π, we reach step 9. Note that $\mu[x_3 \mapsto 1] = \neg y_1 \Rightarrow \texttt{Cb1}[3] = \neg y_1$. Adding $\neg y_1$ to $\texttt{r1}[3]$ makes it $\{\neg y_1\}$. Since $\texttt{r0}[3] = \neg y_2$, and π does not satisfy $\texttt{r0}[3]$, we exit from the loop.

If $x_l = 0$ in π, a similar argument as above shows that $\mu[x_l \mapsto 0]$ can be added to $\texttt{r0}[l]$. Steps 17–18 of *UpdateAbsRef* implement this update. As before, it is

easy to see that $\pi \models \mathtt{r0}[l]$ after step 18. Moreover, since $\pi \models \bigwedge_{i=1}^{n}(x_i \Leftrightarrow \psi_i^A)$ and $\psi_i^A \equiv \neg\mathtt{r1}[l]$, in order to have $x_l = 0$ in π, we must have $\pi \models \mathtt{r1}[l]$. Therefore, we have once again found an index $l > k$ such that $\pi \models \mathtt{r0}[k] \wedge \mathtt{r1}[k]$, and can repeat our algorithm starting with l instead of k. Steps 19–21 of algorithm *UpdateAbsRef* effectively implement this.

Once we exit the loop in steps 6–21 of *UpdateAbsRef*, we compute the refined Skolem function vector Ψ^{A} as $(\neg\mathtt{r1}[1], \ldots \neg\mathtt{r1}[n])$ in step 22 and return the updated $\mathtt{r0}[i]$, $\mathtt{r1}[i]$ for all x_i in X, and also Ψ^{A}.

Example (Continued): Recall that, we changed $\mathtt{r1}[2]$ to $\neg y_1 \vee \neg x_3$, and $\mathtt{r1}[3]$ to $\neg y_1$. Thus after exit from the loop, we have, $\mathtt{r1}[1] = \neg y_1 \vee \neg x_2$, $\mathtt{r1}[2] = \neg y_1 \vee \neg x_3$, and $\mathtt{r1}[3] = \neg y_1$. The refined Skolem function vector Ψ^{A} is $(y_1 \wedge x_2, y_1 \wedge x_3, y_1)$. Note that $\mathtt{r1}[2]$ and $\mathtt{r1}[3]$ have become more abstract after refinement, which has led to more refined ψ_2^A and ψ_3^A.

We now present the proof of Lemma 10.

Proof of Lemma 10. Part (a): Consider an assignment π' of variables in $X \cup Y$, such that $\pi'(x_i) = \pi(x_i')$ for all $x_i \in X$, and $\pi'(y_j) = \pi(y_j)$ for all $y_j \in Y$. Since $\pi \models \varepsilon$, by definition of ε, we have $\pi \models F(X', Y)$. This implies that $\pi' \models F(X, Y)$ and hence, $\pi' \models \exists x_1. \ldots \exists x_{n-1}. F$. If $x_n = 1$ in π', we get $\pi' \models (\exists x_1. \ldots \exists x_{n-1}. F)\,[x_n \mapsto 1]$, or equivalently, $\pi' \models \neg\mathtt{Cb1}[n]$. If $x_n = 0$ in π', by a similar argument, $\pi' \models \neg\mathtt{Cb0}[n]$. Therefore, $\pi' \models \neg\mathtt{Cb1}[n] \vee \neg\mathtt{Cb0}[n]$. Since x_n is the variable with the highest index in X, both $\mathtt{Cb1}[n]$ and $\mathtt{Cb0}[n]$ have only Y as their support. Since $\pi'(y_j) = \pi(y_j)$ for all $y_j \in Y$, it follows that $\pi \models \neg\mathtt{Cb1}[n] \vee \neg\mathtt{Cb0}[n]$ as well.

Part (b): Since $\pi \models \varepsilon$, by definition of ε, we have $\pi \models \neg F(X, Y)$. Since $F = \bigwedge_{q=1}^{r} f^q$, there exists $j \in \{1, \ldots r\}$ such that $\pi \models \neg f^j$. Without loss of generality, assume that $\mathsf{Supp}(f^j) \neq \emptyset$ (otherwise, f^j can be removed from $\bigwedge_{q=1}^{r} f^q$). Let x_k

be the variable with the smallest index in $\mathsf{Supp}(f^j)$. We claim that $x_k = 0$ in π, and prove this by contradiction.

If possible, let $x_k = 1$ in π. Then, $\pi \models (\neg f^j)[x_k \mapsto 1]$. Since x_k is the lowest indexed variable in $\mathsf{Supp}(f^j)$, it follows from algorithm *InitAbsRef* that $(\neg f^j)[x_k \mapsto 1] \in \mathtt{r1}[k]_{init}$, when $\mathtt{r1}[k]_{init}$ is viewed as a set. This implies that $(\neg f^j)[x_k \mapsto 1] \Rightarrow \mathtt{r1}[k]_{init}$, when $\mathtt{r1}[k]_{init}$ is viewed as a function. Hence, $\pi \models \mathtt{r1}[k]_{init}$, and since $\mathtt{r1}[k]_{init} \Rightarrow \mathtt{r1}[k]$, we have $\pi \models \mathtt{r1}[k]$. By definition of ε, we also have $\pi \models (x_k \Leftrightarrow \psi_k^A)$, where $\psi_k^A = \neg\mathtt{r1}[k]$. It follows that $x_k = \psi_k^A = 0$ in π. This contradicts our assumption ($x_k = 1$), and hence x_k must be 0 in π.

Since $x_k = 0$ in π, following the same reasoning as above, we can show that $\pi \models \mathtt{r0}[k]$. Furthermore, since $\pi \models (x_k \Leftrightarrow \psi_k^A)$ and $\psi_k^A = \neg\mathtt{r1}[k]$, having $x_k = 0$ in π implies that $\pi \models \mathtt{r1}[k]$. Hence, $\pi \models \mathtt{r0}[k] \wedge \mathtt{r1}[k]$. It now follows from part (a) that $k \neq n$ and hence $k \in \{1, \ldots n-1\}$

Part (c): We prove this by contradiction. If possible, let there be a Skolem function vector Ψ such that $\psi_i \Leftrightarrow \psi_i^A$ for all i in $\{k+1, \ldots n\}$. Since $\pi \models F(X', Y)$, it follows that $\pi \models \exists x_1. \ldots \exists x_n. F$. Therefore, by definition of Skolem functions, $\pi \models (\cdots(F[x_1 \mapsto \psi_1])\cdots[x_n \mapsto \psi_n])$. Since we have assumed $\psi_i \Leftrightarrow \psi_i^A$ for all i in $\{k+1, \ldots n\}$ and since $\pi \models \bigwedge_{i=1}^{n}(x_i \Leftrightarrow \psi_i^A)$, it follows that $\pi \models (\cdots(F[x_1 \mapsto \psi_1]) \cdots [x_k \mapsto \psi_k])$. However, we know from part (b) that $\pi \models \mathtt{r0}[k] \wedge \mathtt{r1}[k]$ and hence $\pi \models \mathtt{Cb0}[k] \wedge \mathtt{Cb1}[k]$. Recalling the definitions of $\mathtt{Cb0}[k]$ and $\mathtt{Cb1}[k]$, we get $\pi \models (\neg\exists x_1. \ldots \exists x_k. F)$. This contradicts our inference above, i.e. $\pi \models (\cdots(F[x_1 \mapsto \psi_1])\cdots[x_k \mapsto \psi_k])$. Hence our assumption is wrong, i.e. there is no Skolem function vector Ψ such that $\psi_i \Leftrightarrow \psi_i^A$ for all i in $\{k+1, \ldots n\}$.

Part (d): We prove this by contradiction. If possible, suppose $x_l = 0$ in π, or $\pi \models \neg\mathtt{Cb1}[l] \vee \mathtt{r0}[l]$ for all $l \in \{k+1, \ldots n\}$. For convenience of notation, let us

call this assumption A in the discussion below.

If $x_l = 0$ in π, then since $\pi \models \bigwedge_{i=1}^{n}(x_i \Leftrightarrow \psi_i^A)$ and $\psi_i^A = \neg \mathtt{r1}[i]$ for all $i \in \{1,\ldots n\}$, it follows that $\pi \models \mathtt{r1}[l]$. Since $\mathtt{r1}[l] \Rightarrow \mathtt{Cb1}[l]$, we have $\pi \models \mathtt{Cb1}[l]$ as well. It is also easy to see that whenever $\pi \models \neg\mathtt{Cb1}[l]$, then $\pi \models \neg\mathtt{r1}[l]$ as well. Therefore, if $x_l = 0$ in π or if $\pi \models \neg\mathtt{Cb1}[l]$, then both $\mathtt{Cb1}[l]$ and $\mathtt{r1}[l]$ evaluate to the same value under π.

Consider the subcase of assumption A where $x_l = 0$ in π, or $\pi \models \neg\mathtt{Cb1}[l]$, for all $l \in \{k+1,\ldots n\}$. From the discussion above, either $\pi \models \mathtt{Cb1}[l] \wedge \mathtt{r1}[l]$ or $\pi \models \neg\mathtt{Cb1}[l] \wedge \neg\mathtt{r1}[l]$ for all $l \in \{k+1,\ldots n\}$. Now consider the Skolem function vector Ψ given by Proposition 9. Since $\psi_l = \neg\mathtt{Cb1}[l]$ and $\psi_i^A = \neg\mathtt{r1}[l]$, it follows that there exists a Skolem function vector, viz. Ψ, such that $\psi_l \Leftrightarrow \psi_l^A$ for all l in $\{k+1,\ldots n\}$. This contradicts the assertion in part (c) above. Hence we cannot have $x_l = 0$ in π or $\pi \models \neg\mathtt{Cb1}[l]$, for all $l \in \{k+1,\ldots n\}$.

If assumption A has to hold, there must therefore exist some $l \in \{k+1,\ldots n\}$ such that $x_l = 1$ in π and $\pi \models \mathtt{Cb1}[l] \wedge \mathtt{r0}[l]$. Since $\mathtt{r0}[l] \Rightarrow \mathtt{Cb0}[l]$, we must have $\pi \models \mathtt{Cb1}[l] \wedge \mathtt{Cb0}[l]$ in this case. From part (a), we know that $\pi \models \neg\mathtt{Cb0}[n] \vee \neg\mathtt{Cb1}[n]$. It follows that l is strictly less than n, and we can repeat the entire argument above with assumption A restricted to indices in $\{l+1,\ldots n\}$. Note that $\{l+1,\ldots n\}$ is non-empty (since $l < n$), and is a strict subset of $\{k+1,\ldots n\}$ (since $l \in \{k+1,\ldots n\}$). Therefore, restricting assumption A to smaller subsets of indices can only be done finitely many times, after which there won't be any l in the set of indices under consideration such that $x_l = 1$ in π and $\pi \models \mathtt{Cb1}[l] \wedge \mathtt{r0}[l]$. This shows that assumption A is false, thereby proving the assertion in part (d). □

Lemma 11. *Algorithm* UpdateAbsRef *always terminates, and renders at least*

one $\mathtt{r1}[i]$ *strictly abstract, and at least one* ψ_i^A *strictly refined, for* $i \in \{1,\ldots n\}$.

Proof of Lemma 11. By Lemma 10a, we know that $\pi \models \neg\mathtt{Cb0}[n] \vee \neg\mathtt{Cb1}[n]$, and therefore $\pi \models \neg\mathtt{r0}[n] \vee \neg\mathtt{r1}[n]$. Since steps 12–13 or 17-20 of *UpdateAbsRef* can be executed only when $\pi \models \mathtt{r0}[l] \wedge \mathtt{r1}[l]$, and since l is incremented in every iteration of the loop in steps 6–21, it follows that steps 14–15 must be executed for some $l \leq n$. Therefore, algorithm *UpdateAbsRef* always terminates.

It is easy to see from the pseudo-code of algorithm *UpdateAbsRef* that steps 7–10 and 14-15 must be executed before exiting the while loop (steps 6–21) and terminating. Before executing step 10, we have $x_l = 1$ in π and $\pi \models \bigwedge_{i=1}^{n}(x_i \Leftrightarrow \psi_i^A)$. Since $\psi_l^A \equiv \neg\mathtt{r1}[l]$ before step 10, with $x_l = 1$ in π, it must be the case that $\pi \models \neg\mathtt{r1}[l]$ before step 10. However, since $\pi \models \mu[x_l \mapsto 1]$ in step 9, we have $\pi \models \mathtt{r1}[l]$ after step 10. Therefore, executing step 10 renders $\mathtt{r1}[l]$ strictly abstract than what it was earlier. This also implies that $\psi_l^A \equiv \neg\mathtt{r1}[l]$ is strictly refined when *UpdateAbsRef* returns in step 23. □

The *CegarSkolem* algorithm can now be implemented as shown in Algorithm 15.

Theorem 2. CegarSkolem*(*$F(X,Y)$*) terminates and computes a Skolem function vector for* X *in* F

The proof of this follows from Lemmas 8, 9 and 11.

Proof of Theorem 2. By Lemma 11, we know that every invocation of *UpdateAbsRef* renders at least one $\mathtt{r1}[i]$ strictly abstract than what it was earlier. Since $\mathtt{r1}[i]$ is a propositional function, it has finitely many minterms and can be rendered strictly abstract only finitely many times. From Proposition 9, we also know

Algorithm 15: *CegarSkolem*

Input: Propositional formula $F(X,Y) = \bigwedge_{j=1}^{r} f^j(X_j, Y_j)$, where

$X = (x_1, \ldots x_n)$

Output: Skolem function vector $\Psi(Y)$ for X in F

1 $(\Psi^{\mathsf{A}}, \{\mathtt{r0}[i], \mathtt{r1}[i] : 1 \leq i \leq n\}) := \mathit{InitAbsRef}(\bigwedge_{j=1}^{r} f^j)$;

2 $\varepsilon := F(X', Y) \wedge \bigwedge_{i=1}^{n}(x_i \Leftrightarrow \psi_i^A) \wedge \neg F(X, Y)$;

3 **while** ε *is satisfiable* **do**

4 Let π be a satisfying assignment of ε;

5 $(\Psi^{\mathsf{A}}, \{\mathtt{r0}[i], \mathtt{r1}[i] : 1 \leq i \leq n\}) :=$
UpdateAbsRef$(\{\mathtt{r0}[i], \mathtt{r1}[i] : 1 \leq i \leq n\}, \pi)$;

6 $\varepsilon := F(X', Y) \wedge \bigwedge_{i=1}^{n}(x_i \Leftrightarrow \psi_i^A) \wedge \neg F(X, Y)$;

7 $\Psi(Y) := \mathit{ReverseSubstitute}(\neg\mathtt{r1}[1], \ldots \neg\mathtt{r1}[n])$;

8 **return** $\Psi(Y)$;

that $(\neg \texttt{Cb1}[1], \ldots \neg \texttt{Cb1}[n])$ is indeed a Skolem function vector, and therefore by Lemma 9, its error formula is unsatisfiable. The termination of *CegarSkolem* follows immediately from the above observations. Since ε is unsatisfiable when *CegarSkolem* terminates, it follows from Lemma 9 that the vector of functions returned is a Skolem function vector for X in F. □

Example (Continued): As we saw, the refined Skolem function vector Ψ^{A} is $(y_1 \wedge x_2,\ y_1 \wedge x_3,\ y_1)$. Hence, the new error formula is $(y_1 \Leftrightarrow x'_1) \wedge (x'_1 \Leftrightarrow x'_2) \wedge (x'_2 \Leftrightarrow x'_3) \wedge (x'_3 \vee y_2) \wedge (x_1 \Leftrightarrow y_1 \wedge x_2) \wedge (x_2 \Leftrightarrow y_1 \wedge x_3) \wedge (x_3 \Leftrightarrow y_1) \wedge \neg((y_1 \Leftrightarrow x_1) \wedge (x_1 \Leftrightarrow x_2) \wedge (x_2 \Leftrightarrow x_3) \wedge (x_3 \vee y_2))$, which is unsatisfiable. Recall from Section 5.2 that $(y_1 \wedge x_2, y_1 \wedge x_3, y_1)$ is a Skolem function vector for X in F. After *ReverseSubstitute*, we have $(\psi_1, \psi_2, \psi_3) = (y_1, y_1, y_1)$.

5.3.5 Variants

In this subsection, we describe some variants of *CegarSkolem* that we explored. Notice that *CegarSkolem* terminates only when the error formula ε becomes unsatisfiable. Hence the performance of *CegarSkolem* crucially depends on the number of CEGAR iterations and the time consumed in SAT solver calls that check the satisfiability of the error formula ε. The primary motivation behind exploring these variants was to simplify the SAT solver calls and to reduce the number of CEGAR iterations.

Optimization Using Interpolants

Recall that at step 10 of *UpdateAbsRef*, μ_1 is added to $\texttt{r1}[l]$ to obtain a more abstract version of $\texttt{r1}[l]$. Recall that $\mu_1 \Rightarrow \texttt{Cb1}[l]$. Hence $\mu_1 \wedge F(X,Y) \wedge x_l$ is

unsatisfiable. Let ν be an interpolant of μ_1 and $F(X,Y) \wedge x_l$. Note that $\mu_1 \Rightarrow \nu$ and $\nu \wedge F(X,Y) \wedge x_l$ is unsatisfiable. Hence rather than adding μ_1 to $\mathtt{r1}[l]$, we can add ν to $\mathtt{r1}[l]$. Similarly, at step 18 of *UpdateAbsRef*, rather than adding μ_0, we can add an interpolant of μ_0 and $F(X,Y) \wedge \neg x_l$ to $\mathtt{r0}[l]$.

Simplification of ε

The error formula ε in *CegarSkolem* involves two copies of F: $F(X',Y)$ and $\neg F(X,Y)$. It turns out that ε can be simplified by replacing the occurrence of $\neg F(X,Y)$ by a simpler formula.

Definition 3. *The α function for x_i in F, denoted $\alpha[x_i](F)$, is defined to be* $(\exists x_1. \ldots \exists x_{i-1}.F)[x_i \mapsto 1] \wedge (\neg\exists x_1 \ldots \exists x_{i-1}.F)[x_i \mapsto 0]$. *Similarly, the β function for x_i in F, denoted $\beta[x_i](F)$, is defined to be* $(\exists x_1 \ldots \exists x_{i-1}.F)[x_i \mapsto 0] \wedge (\neg\exists x_1. \ldots \exists x_{i-1}.F)[x_i \mapsto 1]$. *When X and F are clear from the context, we use $\alpha[i]$ and $\beta[i]$ for $\alpha[x_i](F)$ and $\beta[x_i](F)$, respectively.*

We change *InitAbsRef* so that it computes refinements of $\alpha[i]$ and $\beta[i]$ denoted as $\widehat{\alpha}[i]$ and $\widehat{\beta}[i]$ (see *InitAbsRef_Mod1*: Algorithm 16). We represent $\widehat{\alpha}[i]$ and $\widehat{\beta}[i]$ as sets of implicitly disjoined functions. Note that *InitAbsRef_Mod1* considers each factor f in $\bigwedge_{j=1}^{r} f^j(X_j,Y_j)$ and accumulates the contribution of f to $\alpha[i]$ in $\widehat{\alpha}[i]$ for every x_i in the support of f. Similarly the contribution of f to $\beta[i]$ are accumulated in $\widehat{\beta}[i]$. If $x_i \in \mathsf{Supp}(f)$, the contribution of f to $\alpha[i]$ is $(\exists x_1 \ldots \exists x_{i-1}.\, f)[x_i \mapsto 1] \wedge (\neg\exists x_1 \ldots \exists x_{i-1}.\, f)[x_i \mapsto 0]$, and its contribution to $\beta[i]$ is $(\exists x_1 \ldots \exists x_{i-1}.\, f)[x_i \mapsto 0] \wedge (\neg\exists x_1 \ldots \exists x_{i-1}.\, f)[x_i \mapsto 1]$.

InitAbsRef_Mod1 also computes $\widehat{\alpha}[i] \wedge \widehat{\beta}[i]$, which we call as bad_i. We formally define $\widehat{\alpha}[i]$, $\widehat{\beta}[i]$, and bad_i as follows.

Definition 4. *The* $\widehat{\alpha}$ *function for* x_i *in* $F = \bigwedge_{j=1}^{r} f^j$*, denoted* $\widehat{\alpha}[x_i](F)$*, is defined to be* $\bigvee_{j=1}^{r} \left(\left(\exists x_1 \ldots \exists x_{i-1}.\ f^j\right)[x_i \mapsto 1] \wedge \left(\neg\exists x_1 \ldots \exists x_{i-1}.\ f^j\right)[x_i \mapsto 0]\right)$*. Similarly the* $\widehat{\beta}$ *function for* x_i *in* F*, denoted* $\widehat{\beta}[x_i](F)$*, is defined to be* $\bigvee_{j=1}^{r}(\left(\exists x_1 \ldots \exists x_{i-1}.\ f^j\right)[x_i \mapsto 0] \wedge \left(\neg\exists x_1 \ldots \exists x_{i-1}.\ f^j\right)[x_i \mapsto 1])$*. When* X *and* F *are clear from the context, we use* $\widehat{\alpha}[i]$ *and* $\widehat{\beta}[i]$ *for* $\widehat{\alpha}[x_i](F)$ *and* $\widehat{\beta}[x_i](F)$*, respectively.*

Definition 5. *The* bad *function for* x_i *in* $F = \bigwedge_{j=1}^{r} f^j$*, denoted* $\text{bad}_{x_i}(F)$*, is defined to be* $\widehat{\alpha}[x_i](F) \wedge \widehat{\beta}[x_i](F)$*. When* X *and* F *are clear from the context, we use* bad_i *for* $\text{bad}_{x_i}(F)$*.*

Notice that *InitAbsRef_Mod1* computes ψ_i^A as $\neg\mathtt{r1}[i] \vee \mathtt{r0}[i]$ unlike *InitAbsRef* which computes ψ_i^A as $\neg\mathtt{r1}[i]$. Similarly, we change *UpdateAbsRef* so that it computes Ψ^{A} as $(\neg\mathtt{r1}[1] \vee \mathtt{r0}[1]_{init}, \ldots \neg\mathtt{r1}[n] \vee \mathtt{r0}[n]_{init})$, where $\mathtt{r0}[i]_{init}$ denotes $\mathtt{r0}[i]$ as computed by *InitAbsRef_Mod1*. The changed version of *UpdateAbsRef* is called *UpdateAbsRef_Mod1*. As we will see, these changes are important for applying the simplification on ε that we describe here.

Since $\mathtt{r0}[i]_{init} \Rightarrow \mathtt{Cb0}[i]$, for any valuation of variables $(x_{i+1}, \ldots x_n)$ and Y, if $\mathtt{r0}[i]_{init}$ evaluates to 1, then any Skolem function for x_i in F should also evaluate to 1. Hence using $\neg\mathtt{r1}[i] \vee \mathtt{r0}[i]_{init}$ instead of $\neg\mathtt{r1}[i]$ as ψ_i^A does not cause any loss of correctness. Thus Theorem 2 can be proved on the variant of *CegarSkolem* that calls *InitAbsRef_Mod1* and *UpdateAbsRef_Mod1* in place of *InitAbsRef* and *UpdateAbsRef*.

More interestingly, these changes allow us to use a different version of the error formula.

Lemma 12. *The formula* ε' *defined as* $F(X', Y) \wedge \bigwedge_{i=1}^{n}(x_i \Leftrightarrow \psi_i^A) \wedge (\text{bad}_1 \vee \ldots \vee \text{bad}_{n-1})$ *for* Ψ^{A} *computed as* $(\neg\mathtt{r1}[1] \vee \mathtt{r0}[1]_{init}, \ldots \neg\mathtt{r1}[n] \vee \mathtt{r0}[n]_{init})$ *is unsatisfiable iff* Ψ^{A} *is a Skolem function vector for* X *in* F*.*

Algorithm 16: *InitAbsRef_Mod1*

Input: Prop. formula $F(X,Y) = \bigwedge_{j=1}^{r} f^j(X_j,Y_j)$, where $X = (x_1,\ldots x_n)$

Output: Abstract Skolem function vector $\Psi^{\mathsf{A}} = (\psi_1^A,\ldots,\psi_n^A)$, $\mathtt{r0}[i]$, $\mathtt{r1}[i]$, $\widehat{\alpha}[i]$, $\widehat{\beta}[i]$, and bad_i for each x_i in X

1 **for** *i in* 1 *to n* **do**

2 $\quad \mathtt{r0}[i] := \emptyset$; $\mathtt{r1}[i] := \emptyset$; $\widehat{\alpha}[i] := \emptyset$; $\widehat{\beta}[i] := \emptyset$; `// Initializing`

3 **for** *j in* 1 *to r* **do**

4 $\quad f := f^j$; `// Consider factor separately`

5 $\quad$ **for** *i in* 1 *to n* **do**

6 $\quad\quad$ **if** $x_i \in \mathsf{Supp}(f)$ **then**

7 $\quad\quad\quad \mathtt{r0}[i] := \mathtt{r0}[i] \cup \{\neg f[x_i \mapsto 0]\}$;

8 $\quad\quad\quad \mathtt{r1}[i] := \mathtt{r1}[i] \cup \{\neg f[x_i \mapsto 1]\}$;

9 $\quad\quad\quad \widehat{\alpha}[i] := \widehat{\alpha}[i] \cup \{f[x_i \mapsto 1] \wedge \neg f[x_i \mapsto 0]\}$;

10 $\quad\quad\quad \widehat{\beta}[i] := \widehat{\beta}[i] \cup \{f[x_i \mapsto 0] \wedge \neg f[x_i \mapsto 1]\}$;

$\quad\quad\quad$ `// Skolem function for` x_i `in` f

11 $\quad\quad\quad \psi_{i,f} := f[x_i \mapsto 1]$;

12 $\quad\quad\quad f := f[x_i \mapsto \psi_{i,f}]$; `//` $f[x_i \mapsto \psi_{i,f}] \equiv \exists x_i.\, f$

13 **for** *i in* 1 *to n* **do**

14 $\quad \psi_i^A := \neg \mathtt{r1}[i] \vee \mathtt{r0}[i]$;

15 $\quad \mathrm{bad}_i := \widehat{\alpha}[i] \wedge \widehat{\beta}[i]$;

$\quad$ `// Interpreting` $\mathtt{r1}[i]$, $\mathtt{r0}[i]$, $\widehat{\alpha}[i]$, $\widehat{\beta}[i]$ `as functions`

16 **return** $\Psi^{\mathsf{A}} = (\psi_1^A,\ldots\psi_n^A)$, $\mathtt{r0}[i], \mathtt{r1}[i]$, $\widehat{\alpha}[i]$, $\widehat{\beta}[i]$, *and* bad_i *for each* $x_i \in X$

Proof of Lemma 12. Proof in one direction is easy. Let us assume that $\psi_1^A, \ldots, \psi_n^A$ are Skolem functions for variables $x_1, \ldots, x_n$ in F. Let us consider any solution π of $F(X', Y)$, i.e., $F(x'_1, \ldots, x'_n, Y)$. Let v_Y be the value assigned to the variables in Y by π. Let $\psi_1^A, \ldots, \psi_n^A$ evaluate to $v_1, \ldots, v_n$ for $Y = v_Y$. Note that $F(x_1, \ldots, x_n, Y)$ $[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]$ $[Y \mapsto v_Y]$ is true, since $\psi_1^A, \ldots, \psi_n^A$ are Skolem functions for variables $x_1, \ldots, x_n$ in F. Suppose some bad_j is true for $(x_1 = v_1)$, $\ldots$, $(x_n = v_n)$, $(Y = v_Y)$, where $1 \leq j \leq n-1$. This means that both $\widehat{\alpha}[j]$ and $\widehat{\beta}[j]$ are true, which actually implies that one of the factors will evaluate to false for $(x_1 = v_1)$, $\ldots$, $(x_n = v_n)$, $(Y = v_Y)$, i.e., F will evaluate to false for $(x_1 = v_1)$, $\ldots$, $(x_n = v_n)$, $(Y = v_Y)$. This means that $\text{bad}_1, \ldots, \text{bad}_{n-1}$ should be false for $(x_1 = v_1)$, $\ldots$, $(x_n = v_n)$, $(Y = v_Y)$. Hence, if $\psi_1^A, \ldots, \psi_n^A$ are Skolem functions for variables $x_1, \ldots, x_n$ in F, then for any value v_Y of variables in Y for which there exists $x'_1, \ldots, x'_n$ such that $F(x'_1, \ldots, x'_n, Y)$ is satisfiable, $\text{bad}_1, \ldots, \text{bad}_{n-1}$ should be false, and hence ε' should be false. This means that ε' is unsatisfiable.

Let us consider the proof in the other direction.

In the nested *for* loop in Algorithm 16 steps 3–12, we make an additional assumption that $x_i \in \mathsf{Supp}(f)$ is always true. This assumption is only to simplify the notation, and the proof can be done even without this assumption. We denote $\exists x_1 \ldots \exists x_i.\, f^1$, …, $\exists x_1 \ldots \exists x_i.\, f^r$ as $f_i^1, \ldots, f_i^r$. The factors $f^1, \ldots, f^r$ are denoted as $f_0^1, \ldots, f_0^r$. We define ω_i, for $2 \leq i \leq n$, as $(\text{bad}_i \vee (\widehat{\alpha}[i] \wedge \omega_{i-1}[x_i \mapsto 1]) \vee (\widehat{\beta}[i] \wedge \omega_{i-1}[x_i \mapsto 0]) \vee (\omega_{i-1}[x_i \mapsto 1] \wedge \omega_{i-1}[x_i \mapsto 0]))$, and ω_1 as bad_1.

Let us assume that ε' is unsatisfiable. Note that if $F(x'_1, \ldots, x'_n, Y)$ is unsatisfiable, then any $\psi_1^A, \ldots, \psi_n^A$ are Skolem functions for variables $x_1, \ldots, x_n$ in F.

The interesting case is when $F(x'_1, \ldots, x'_n, Y)$ is satisfiable. This implies that for any value v_Y of variables in Y such that $F(x'_1, \ldots, x'_n, Y)$ is satisfiable, $\text{bad}_1, \ldots, \text{bad}_{n-1}$

are false. In the following, we will prove that in this case, $\psi_1^A, \ldots, \psi_n^A$ are Skolem functions for variables $x_1, \ldots, x_n$ in F. Our proof makes use of the following claims.

Claim 3. *ψ_i^A is a Skolem function for x_i in $f_{i-1}^1 \wedge \ldots \wedge f_{i-1}^r$.*

Claim 4. *$F(x_1, \ldots, x_n, Y)\ [x_1 \mapsto \psi_1^A] \ldots [x_n \mapsto \psi_n^A]$ is equivalent to $(f_n^1 \wedge \ldots \wedge f_n^r)$ $\wedge\ \neg \mathrm{bad}_1[x_2 \mapsto \psi_2^A] \ldots [x_n \mapsto \psi_n^A] \wedge \ldots \wedge \neg \mathrm{bad}_{n-1}[x_n \mapsto \psi_n^A] \wedge \neg \mathrm{bad}_n$.*

Claim 5. *$\exists x_1 \ldots . \exists x_n . F(x_1, \ldots, x_n, Y)$ is equivalent to $(f_n^1 \wedge \ldots \wedge f_n^r) \wedge \neg \omega_n$.*

Claim 6. *For any value v_Y of variables in Y such that $F(x_1', \ldots, x_n', Y)$ is satisfiable, and values of $x_1, \ldots, x_n$ such that $(x_1 = \psi_1^A), \ldots, (x_n = \psi_n^A)$ for which $\mathrm{bad}_1, \ldots, \mathrm{bad}_{n-1}$ are* false*, ω_n evaluates to* false*.*

Proof of Claim 3. Recall that the Skolem function ψ_i^A has the form $\neg \mathtt{r1}[i] \vee \mathtt{r0}[i]_{init}$. Hence,

- Case 1: For the values of the other variables for which x_i must be set to true for $f_{i-1}^1 \wedge \ldots \wedge f_{i-1}^r$ to become true, $\widehat{\alpha}[i]$ is true. Hence there exists an f_{i-1}^j such that $\neg f_{i-1}^j[x_i \mapsto 1]$ is false and $\neg f_{i-1}^j[x_i \mapsto 0]$ is true. Since $\neg f_{i-1}^j[x_i \mapsto 0] \Rightarrow \mathtt{r0}[i]_{init}$, ψ_i^A is true.

- Case 2: For the values of the other variables for which x_i must be set to false for $f_{i-1}^1 \wedge \ldots \wedge f_{i-1}^r$ to become true, $\widehat{\beta}[i]$ is true. Hence there exists an f_{i-1}^j such that $\neg f_{i-1}^j[x_i \mapsto 1]$ is true and $\neg f_{i-1}^j[x_i \mapsto 0]$ is false. Since $\neg f_{i-1}^j[x_i \mapsto 1] \Rightarrow \mathtt{r1}[i]$, we have $\mathtt{r1}[i] =$ true and $\neg \mathtt{r1}[i] =$ false. Moreover, since $f_{i-1}^1 \wedge \ldots \wedge f_{i-1}^r$ is true, there cannot exist any f_{i-1}^k such that (i) $\neg f_{i-1}^k[x_i \mapsto 1]$ is true and $\neg f_{i-1}^k[x_i \mapsto 0]$ is true or (ii) $\neg f_{i-1}^k[x_i \mapsto 1]$ is false and $\neg f_{i-1}^k[x_i \mapsto 0]$

is true. This means that for all f_{i-1}^k, we have $\neg f_{i-1}^k[x_i \mapsto 0]$ as false. Hence $\mathtt{r0}[i]_{init}$ is false, and hence ψ_i^A is false.

Similarly, it can be observed that

– Case 3: For the values of the other variables for which x_i can be set to true or false for $f_{i-1}^1 \wedge \ldots \wedge f_{i-1}^r$ to become true, ψ_i^A is true.

– Case 4: For the values of the other variables for which one of the factors in $f_{i-1}^1 \wedge \ldots \wedge f_{i-1}^r$ cannot be satisfied, ψ_i^A is $\widehat{\alpha}[i]$.

– Case 5: For the values of the other variables for which $\widehat{\alpha}[i]$ is true and $\widehat{\beta}[i]$ is true, ψ_i^A is true.

Note that in each of these cases, $(f_{i-1}^1 \wedge \ldots \wedge f_{i-1}^r)\,[x_i \mapsto \psi_i^A]$ is equivalent to $\exists x_i.\,(f_{i-1}^1 \wedge \ldots \wedge f_{i-1}^r)$, and hence ψ_i^A is a Skolem function for x_i in $f_{i-1}^1 \wedge \ldots \wedge f_{i-1}^r$.

(end of proof of Claim 3) □

Proof of Claim 4. From Claim 3, $F(x_1,\ldots,x_n,Y)\,[x_1 \mapsto \psi_1^A]$ is equivalent to $\exists x_1.\,(f_0^1 \wedge \ldots \wedge f_0^r)$. Note that $\exists x_1.\,(f_0^1 \wedge \ldots \wedge f_0^r)$ is equivalent to $(f_1^1 \wedge \ldots \wedge f_1^r) \wedge \neg \mathrm{bad}_1$.

Similarly, $F(x_1,\ldots,x_n,Y)\,[x_1 \mapsto \psi_1^A]\,[x_2 \mapsto \psi_2^A]$ is equivalent to $((f_1^1 \wedge \ldots \wedge f_1^r) \wedge \neg \mathrm{bad}_1)\,[x_2 \mapsto \psi_2^A]$. Note that $(f_1^1 \wedge \ldots \wedge f_1^r)\,[x_2 \mapsto \psi_2^A]$ is equivalent to $(f_2^1 \wedge \ldots \wedge f_2^r) \wedge \neg \mathrm{bad}_2$. Hence $F(x_1,\ldots,x_n,Y)\,[x_1 \mapsto \psi_1^A]\,[x_2 \mapsto \psi_2^A]$ is equivalent to $(f_2^1 \wedge \ldots \wedge f_2^r) \wedge \neg \mathrm{bad}_2 \wedge \neg \mathrm{bad}_1[x_2 \mapsto \psi_2^A]$.

Proceeding in this manner, it can be proved that $F(x_1,\ldots,x_n,Y)\,[x_1 \mapsto \psi_1^A] \ldots [x_n \mapsto \psi_n^A]$ is equivalent to $(f_n^1 \wedge \ldots \wedge f_n^r) \wedge \neg \mathrm{bad}_1[x_2 \mapsto \psi_2^A] \ldots [x_n \mapsto \psi_n^A] \wedge \ldots \wedge \neg \mathrm{bad}_{n-1}[x_n \mapsto \psi_n^A] \wedge \neg \mathrm{bad}_n$. (end of proof of Claim 4) □

Proof of Claim 5. We will prove that $\exists x_1 \ldots \exists x_i.\,F(x_1,\ldots,x_n,Y)$ is equivalent to $(f_i^1 \wedge \ldots \wedge f_i^r) \wedge \neg \omega_i$ for $1 \leq i \leq n$ using induction on i.

Base case is easy to prove. Notice that $\exists x_1.F(x_1,\ldots,x_n,Y)$ is equivalent to $(f_1^1 \wedge \ldots \wedge f_1^r) \wedge \neg \text{bad}_1$. Note that bad_1 is the same as ω_1. Hence, $\exists x_1.F(x_1,\ldots,x_n,Y)$ is equivalent to $(f_1^1 \wedge \ldots \wedge f_1^r) \wedge \neg \omega_1$.

Suppose $\exists x_1 \ldots. \exists x_{i-1}.F(x_1,\ldots,x_n,Y)$ is equivalent to $(f_{i-1}^1 \wedge \ldots \wedge f_{i-1}^r) \wedge \neg \omega_{i-1}$. Note that $\exists x_1 \ldots. \exists x_i.F(x_1,\ldots,x_n,Y)$ is $\exists x_i.((f_{i-1}^1 \wedge \ldots \wedge f_{i-1}^r) \wedge \neg \omega_{i-1})$. It can be observed that this is equivalent to $(f_i^1 \wedge \ldots \wedge f_i^r) \wedge \neg\ (\text{bad}_i \vee (\widehat{\alpha}[i] \wedge \omega_{i-1}[x_i \mapsto 1]) \vee (\widehat{\beta}[i] \wedge \omega_{i-1}[x_i \mapsto 0]) \vee (\omega_{i-1}[x_i \mapsto 1] \wedge \omega_{i-1}[x_i \mapsto 0]))$, which is the same as $(f_i^1 \wedge \ldots \wedge f_i^r) \wedge \neg \omega_i$. (end of proof of Claim 5) □

Proof of Claim 6. Consider any value v_Y of variables in Y such that $F(x_1',\ldots,x_n',Y)$ is satisfiable, and values of $x_1,\ldots,x_n$ such that $(x_1 = \psi_1^A),\ldots,(x_n = \psi_n^A)$ for which $\text{bad}_1,\ldots,\text{bad}_{n-1}$ are false. Let $\psi_1^A,\ldots,\psi_n^A$ evaluate to $v_1,\ldots,v_n$ under v_Y. This means that $\text{bad}_{n-1}[x_n \mapsto v_n][Y \mapsto v_Y]$ is false, $\text{bad}_{n-2}[x_{n-1} \mapsto v_{n-1}][x_n \mapsto v_n][Y \mapsto v_Y]$ is false, …, and $\text{bad}_1[x_2 \mapsto v_2]\ \ldots\ [x_n \mapsto v_n]\ [Y \mapsto v_Y]$ is false. Note that $\text{bad}_n[Y \mapsto v_Y]$ is also false, since $F(x_1',\ldots,x_n',Y)$ is satisfiable for $Y = v_Y$.

Recall that ω_i is $\text{bad}_i \vee (\widehat{\alpha}[i] \wedge \omega_{i-1}[x_i \mapsto 1]) \vee (\widehat{\beta}[i] \wedge \omega_{i-1}[x_i \mapsto 0]) \vee (\omega_{i-1}[x_i \mapsto 1] \wedge \omega_{i-1}[x_i \mapsto 0])$. Note that ω_i is a formula on variables $x_{i+1},\ldots,x_n,Y$. Let us evaluate each disjunct in ω_i for $(x_{i+1} = v_{i+1}),\ldots,(x_n = v_n)$, $(Y = v_Y)$.

- Consider bad_i. We have $\text{bad}_i[x_{i+1} \mapsto v_{i+1}]\ \ldots\ [x_n \mapsto v_n]\ [Y \mapsto v_Y]$ is false.
- Consider $\widehat{\alpha}[i] \wedge \omega_{i-1}[x_i \mapsto 1]$. Let $\widehat{\alpha}[i]$ be true for $(x_{i+1} = v_{i+1}),\ldots,(x_n = v_n)$, $(Y = v_Y)$. From Claim 3, we know that ψ_i^A is a Skolem function for x_i in $f_{i-1}^1 \wedge \ldots \wedge f_{i-1}^r$. Hence, if $\widehat{\alpha}[i]$ is true, then ψ_i^A is true, i.e., v_i is true. Therefore, if $\widehat{\alpha}[i]$ is true, then $\widehat{\alpha}[i] \wedge \omega_{i-1}[x_i \mapsto 1]$ is the same as $\omega_{i-1}[x_i \mapsto v_i]$ for $(x_{i+1} = v_{i+1}),\ldots,(x_n = v_n)$, $(Y = v_Y)$.

– Consider $\widehat{\beta}[i] \wedge \omega_{i-1}[x_i \mapsto 0]$. Let $\widehat{\beta}[i]$ be true for for $(x_{i+1} = v_{i+1}), \ldots, (x_n = v_n)$, $(Y = v_Y)$. From Claim 3, we know that ψ_i^A is a Skolem function for x_i in $f_{i-1}^1 \wedge \ldots \wedge f_{i-1}^r$. Hence, if $\widehat{\beta}[i]$ is true, then ψ_i^A is false, i.e., v_i is false. Therefore, if $\widehat{\beta}[i]$ is true, then $\widehat{\beta}[i] \wedge \omega_{i-1}[x_i \mapsto 0]$ is the same as $\omega_{i-1}[x_i \mapsto v_i]$ for $(x_{i+1} = v_{i+1}), \ldots, (x_n = v_n)$, $(Y = v_Y)$.

– Consider $\omega_{i-1}[x_i \mapsto 1] \wedge \omega_{i-1}[x_i \mapsto 0]$. It can be written as $\omega_{i-1}[x_i \mapsto v_i] \wedge \omega_{i-1}[x_i \mapsto \neg v_i]$.

Hence, ω_i reduces to either $\omega_{i-1}[x_i \mapsto v_i]$ or $\omega_{i-1}[x_i \mapsto v_i] \wedge \omega_{i-1}[x_i \mapsto \neg v_i]$ for $(x_{i+1} = v_{i+1}), \ldots, (x_n = v_n)$, $(Y = v_Y)$ where $1 \leq i \leq n$. Since ω_1 is bad_1 and $\text{bad}_1[x_2 \mapsto v_2] \ldots [x_n \mapsto v_n]\ [Y \mapsto v_Y]$ is false, ω_n evaluates to false for $(Y = v_Y)$.

(end of proof of Claim 6) □

Consider any value v_Y of variables in Y such that $F(x'_1, \ldots, x'_n, Y)$ is satisfiable, and values of $x_1, \ldots, x_n$ such that $(x_1 = \psi_1^A), \ldots, (x_n = \psi_n^A)$ for which $\text{bad}_1, \ldots, \text{bad}_{n-1}$ are false. Using Claim 6, ω_n evaluates to false. Since $F(x'_1, \ldots, x'_n, Y)$ is satisfiable for $(Y = v_Y)$, bad_n is false. Now using Claim 4 and Claim 5, both $F(x_1, \ldots, x_n, Y)$ $[x_1 \mapsto \psi_1^A] \ldots [x_n \mapsto \psi_n^A]$ and $\exists x_1 \ldots . \exists x_n . F(x_1, \ldots, x_n, Y)$ evaluate to to the same value $(f_n^1 \wedge \ldots \wedge f_n^r)$. Thus for any value v_Y of variables in Y such that $F(x'_1, \ldots, x'_n, Y)$ is satisfiable, $F(x_1, \ldots, x_n, Y)\ [x_1 \mapsto \psi_1^A] \ldots [x_n \mapsto \psi_n^A]$ iff $\exists x_1 \ldots . \exists x_n . F(x_1, \ldots, x_n, Y)$. Hence $\psi_1^A, \ldots, \psi_n^A$ are Skolem functions for variables $x_1, \ldots, x_n$ in F. □

Note that ε' uses $(\text{bad}_1 \vee \ldots \vee \text{bad}_{n-1})$ in place of $\neg F(X, Y)$. As the bad functions are constructed by considering the individual factors separately without constructing their conjunctions, we expect $(\text{bad}_1 \vee \ldots \vee \text{bad}_{n-1})$ to be a function simpler than $\neg F(X, Y)$. The variant of *CegarSkolem* that uses *InitAbsRef_Mod1*,

UpdateAbsRef_Mod1 and ε' instead of *InitAbsRef, UpdateAbsRef* and ε is called *CegarSkolem_Mod1*.

Refining Skolem Functions Bottom-Up

The error formula ε in *CegarSkolem* checks the correctness of a vector of Skolem functions. Suppose ε is satisfiable. As per Lemma 9, this implies that $\psi_1^A, \ldots \psi_n^A$ are not Skolem functions for $x_1, \ldots x_n$ in F. However, it may happen that $\psi_{k+1}^A, \ldots \psi_n^A$ are already Skolem functions for $x_{k+1}, \ldots x_n$ in F for $1 \leq k < n$, and satisfiability of ε is happening due to the reason that some of the Skolem functions in $\psi_1^A, \ldots \psi_k^A$ are still abstract. In such situations, it is useful to check if a specific Skolem function is correct, rather than checking the correctness of a vector of Skolem functions. We can modify the error formula to achieve this.

Lemma 13. *The formula* ε_k *defined as* $F(X',Y) \wedge \bigwedge_{i=1}^{n}(x_i \Leftrightarrow \psi_i^A) \wedge \neg F(X,Y) \wedge \neg x_k' \wedge x_k \wedge (x_{k+1}' = x_{k+1}) \wedge \ldots \wedge (x_n' = x_n)$ *is unsatisfiable iff* ψ_k^A *is Skolem function for* x_k *in* F*, where* $1 \leq k < n$*.*

Proof of Lemma 13. Note that ε_k is ε with the additional constraints $(x_k = 1)$, $(x_k' = 0)$ and $(x_{k+1}' = x_{k+1})$, ..., $(x_n' = x_n)$. The constraints $(x_{k+1}' = x_{k+1})$, ... $(x_n' = x_n)$ are added under the assumption that $\psi_{k+1}^A, \ldots, \psi_n^A$ are Skolem functions for variables $x_{k+1}, \ldots, x_n$ in F. This is a reasonable assumption to make, since as we will see, we will fix the Skolem functions $\psi_{k+1}^A, \ldots, \psi_n^A$ before fixing ψ_k^A. Since ψ_k^A is an abstract Skolem function, the constraints $(x_k = 1)$ and $(x_k' = 0)$ capture the only condition under which our choice of ψ_k^A is wrong. □

We present a variant of *CegarSkolem* called *CegarSkolem_Mod2* (see Algorithm 17) that fixes the Skolem functions in a pre-defined order using Lemma 13.

Algorithm 17: *CegarSkolem_Mod2*

Input: Propositional formula $F(X,Y) = \bigwedge_{j=1}^{r} f^j(X_j, Y_j)$, where

$X = (x_1, \ldots x_n)$

Output: Skolem function vector $\Psi(Y)$ for X in F

1 $(\Psi^{\mathsf{A}}, \{\mathtt{r0}[i], \mathtt{r1}[i] : 1 \leq i \leq n\}) := \mathit{InitAbsRef}(\bigwedge_{j=1}^{r} f^j)$;

2 **for** *k in n to* 1 **do**

3 $\quad \varepsilon_k := F(X',Y) \wedge \bigwedge_{i=1}^{n}(x_i \Leftrightarrow \psi_i^A) \wedge \neg F(X,Y) \wedge \neg x'_k \wedge x_k \wedge (x'_{k+1} = x_{k+1}) \wedge \ldots \wedge (x'_n = x_n)$;

4 $\quad$ **while** ε_k *is satisfiable* **do**

5 $\quad\quad$ Let π be a satisfying assignment of ε_k;

6 $\quad\quad$ $(\Psi^{\mathsf{A}}, \{\mathtt{r0}[i], \mathtt{r1}[i] : 1 \leq i \leq n\}) :=$ $\mathit{UpdateAbsRef}(\{\mathtt{r0}[i], \mathtt{r1}[i] : 1 \leq i \leq n\}, \pi)$;

7 $\quad\quad$ $\varepsilon_k := F(X',Y) \wedge \bigwedge_{i=1}^{n}(x_i \Leftrightarrow \psi_i^A) \wedge \neg F(X,Y) \wedge \neg x'_k \wedge x_k \wedge (x'_{k+1} = x_{k+1}) \wedge \ldots \wedge (x'_n = x_n)$;

8 $\quad \varepsilon := F(X',Y) \wedge \bigwedge_{i=1}^{n}(x_i \Leftrightarrow \psi_i^A) \wedge \neg F(X,Y)$;

9 $\quad$ **if** ε *is unsatisfiable* **then**

10 $\quad\quad$ **break**;

11 $\Psi(Y) := \mathit{ReverseSubstitute}(\neg\mathtt{r1}[1], \ldots \neg\mathtt{r1}[n])$;

12 **return** $\Psi(Y)$;

CegarSkolem_Mod2 initially checks the satisfiability of ε_n. If ε_n is unsatisfiable, then ψ_n^A is a Skolem function for x_n in F. Otherwise if ε_n is satisfiable, then *CegarSkolem_Mod2* calls *UpdateAbsRef* to refine $\psi_1^A, \ldots \psi_n^A$. Then the satisfiability of ε_n is again checked, and this loop repeats until ψ_n^A becomes a Skolem function for x_n in F. The first iteration of the **for** loop thus fixes Skolem function for x_n. The subsequent iterations fix Skolem functions for variables $x_{n-1}, \ldots, x_1$. After fixing the Skolem function for x_k, the satisfiability of ε is checked to see if $\psi_1^A, \ldots \psi_n^A$ are Skolem functions for $x_1, \ldots x_n$ in F. If ε is unsatisfiable, then *CegarSkolem_Mod2* makes an early exit from the **for** loop.

5.4 Experimental Results

5.4.1 Benchmarks

The Skolem function generation benchmarks were obtained by considering sequential circuits from the HWMCC10 benchmark suite [107], and by formulating the problem of disjunctively decomposing the circuit into components as a problem of generating Skolem function vectors. Each benchmark is of the form $\exists X.F(X,Y)$, where $F(X,Y)$ is a conjunction of factors, and was generated in the following manner.

The HWMCC10 benchmarks are circuits in .aig format. In order to generate our benchmarks, we first read each circuit, and then extracted the symbolic transition function of the circuit. Let $(x'_1 = f_1(X,I)) \wedge \ldots \wedge (x'_n = f_n(X,I))$ be the symbolic transition function extracted, where $X = (x_1, \ldots, x_n)$ is the present state, $X' = (x'_1, \ldots, x'_n)$ is the next state, $I = (i_1, \ldots, i_m)$ are the inputs, and $f_1, \ldots, f_n$ are transition functions for the state variables $x_1, \ldots, x_n$ respectively. We generated

benchmarks of the form $\exists X.F(X,Y)$ from each symbolic transition function in the following manner.

The first benchmark is of the form $\exists I.\ ((x_1 \neq f_1(X,I)) \vee \ldots \vee (x_n \neq f_n(X,I)))$. Note that for a given state X, a value of variables in I that satisfies the formula $(x_1 \neq f_1(X,I)) \vee \ldots \vee (x_n \neq f_n(X,I))$ gives an outgoing edge from X which is not a self-loop. Hence the benchmark represents the problem: *Generate Skolem functions for inputs in I such that the outgoing edge enabled by the chosen values of inputs is not a self-loop*.

To perform factorization, we did one-level of Tseitin encoding [100]. Let $Z = \{Z_1,\ldots,Z_n\}$ be the Tseitin variables introduced. Two versions of the benchmark were generated after Tseitin encoding: (i) $\exists I.((Z_1 = (x_1 \neq f_1(X,I))) \wedge \ldots \wedge (Z_n = (x_n \neq f_n(X,I))) \wedge (Z_1 \vee \ldots \vee Z_n))$, and (ii) $\exists I.\exists Z.((Z_1 = (x_1 \neq f_1(X,I))) \wedge \ldots \wedge (Z_n = (x_n \neq f_n(X,I))) \wedge (Z_1 \vee \ldots \vee Z_n))$.

The second benchmark is of the form $\exists I.((f_1(X,\mathsf{TRUE}) \neq f_1(X,I)) \vee \ldots \vee (f_n(X,\mathsf{TRUE}) \neq f_n(X,I)))$, where TRUE indicates that all inputs are set to true. Note that for a given state X, a value of variables in I that satisfies the formula $(f_1(X,\mathsf{TRUE}) \neq f_1(X,I)) \vee \ldots \vee (f_n(X,\mathsf{TRUE}) \neq f_n(X,I))$ gives an outgoing edge from X which is *not* leading to the same state as led by the edge enabled when values of all inputs are true. Hence the benchmark represents the problem: *Generate Skolem functions for inputs in I such that the outgoing edge enabled by the chosen values of inputs is such that it is not leading to the same state as led by the edge enabled when values of all inputs are* true.

As in the case of the first benchmark, to perform factorization, we introduced Tseitin variables and did one-level of Tseitin encoding. Let $Z = \{Z_1,\ldots,Z_n\}$ be the Tseitin variables introduced. Two versions of the benchmark were generated:

(i) $\exists I.((Z_1 = (f_1(X,\mathsf{TRUE}) \neq f_1(X,I))) \wedge \ldots \wedge (Z_n = (f_n(X,\mathsf{TRUE}) \neq f_n(X,I)))$ $\wedge (Z_1 \vee \ldots \vee Z_n))$, and (ii) $\exists I.\exists Z.((Z_1 = (f_1(X,\mathsf{TRUE}) \neq f_1(X,I))) \wedge \ldots \wedge (Z_n =$ $(f_n(X,\mathsf{TRUE}) \neq f_n(X,I))) \wedge (Z_1 \vee \ldots \vee Z_n))$.

The third benchmark is similar to the first benchmark of the form $\exists I.(x_1 \neq f_1(X,I)) \vee \ldots \vee (x_n \neq f_n(X,I)))$. However, we added a formula $A(X) : (x_1 = f_1(X,g_1(X))) \wedge \ldots \wedge (x_n = f_n(X,g_n(X)))$ as an additional disjunct, where $g_1(X)$, $\ldots, g_n(X)$ are functions of X. Thus we have, $\exists I.((x_1 \neq f_1(X,I)) \vee \ldots \vee (x_n \neq f_n(X,I)) \vee A(X))$. We then did one-level of Tseitin encoding and existentially quantified the Tseitin variables to get the final benchmark as $\exists Z.\exists I.((Z_1 = (x_1 \neq f_1(X,I))) \wedge \ldots \wedge (Z_n = (x_n \neq f_n(X,I))) \wedge (Z_{n+1} = A(X)) \wedge (Z_1 \vee \ldots \vee Z_n \vee Z_{n+1}))$, where $Z = \{Z_1, \ldots, Z_{n+1}\}$ are the Tseitin variables. The additional disjunct $A(X)$ guarantees that $\exists Z.\exists I.((Z_1 = (x_1 \neq f_1(X,I))) \wedge \ldots \wedge (Z_n = (x_n \neq f_n(X,I))) \wedge (Z_{n+1} = A(X)) \wedge (Z_1 \vee \ldots \vee Z_n \vee Z_{n+1}))$ is valid. Note that for any given state X, the variables $Z_1, \ldots, Z_n$ become false when the only outgoing edge from X is a self-loop. However in this case Z_{n+1} is true irrespective of the values of the functions $g_1(X), \ldots, g_n(X)$. Hence $\exists Z.\exists I.((Z_1 = (x_1 \neq f_1(X,I))) \wedge \ldots \wedge (Z_n = (x_n \neq f_n(X,I))) \wedge (Z_{n+1} = A(X)) \wedge (Z_1 \vee \ldots \vee Z_n \vee Z_{n+1}))$ is true for any X.

After generating these benchmarks from the symbolic transition function extracted from each circuit, the benchmarks such that the number of existentially quantified variables is less than 20 are avoided from the benchmark-suite. Similarly the benchmarks for which none of the algorithms used in the experiments could generate Skolem functions are also avoided. Finally we have 424 benchmarks.

We divided the benchmarks into two categories: a) TYPE-1 benchmarks

where $\exists X.F(X,Y)$ is *valid,* and b) TYPE-2 benchmarks where $\exists X.F(X,Y)$ is *invalid*. Among the 424 benchmarks, 160 are TYPE-1 benchmarks and 264 are TYPE-2 benchmarks.

5.4.2 Experimental Methodology

For experimental evaluation, we ran *CegarSkolem* and *MonoSkolem* on all the benchmarks. We also compared the performance of *CegarSkolem* with tools that generate Skolem functions from the proof of validity of $\exists X.F(X,Y)$. Since these tools generate Skolem functions only for valid $\exists X.F(X,Y)$ formulas, and require the input to be in .qdimacs format, we converted each of the TYPE-1 benchmarks into .qdimacs format using Tseitin encoding.

We generated QRAT proofs for the TYPE-1 benchmarks using the bloqqer [108] tool, and then used the qrat-trim tool [17] to generate Skolem functions from the QRAT proofs. We refer to bloqqer + qrat-trim as Bloqqer below. We also generated the cube-resolution proofs in QRP format for the TYPE-1 benchmarks using the QBF solver DepQBF [106], and then used the QBFcert framework [109] to generate Skolem functions from the QRP proofs. We refer to DepQBF + QBFcert as DepQBF below.

Our implementations of *MonoSkolem* and *CegarSkolem* make use of the ABC [99] library to represent and manipulate functions as AIGs. For *CegarSkolem*, we used the default SAT solver provided by ABC, which is a variant of MiniSAT. We used a simple heuristic to order the variables, and used the same ordering for both *MonoSkolem* and *CegarSkolem*. In our ordering, variables that occur in fewer factors are indexed lower than those that occur in more factors.

We used the following metrics to compare the performance of the algorithms: (i) max/average size of the generated Skolem functions in a Skolem function vector, where the size is the number of nodes in the AIG representation of the function, and ii) total time taken to generate the Skolem function vector (excluding any input format conversion time), in seconds. The experiments were performed on a 1.87 GHz Intel(R) Xeon machine with 128GB memory running Ubuntu 12.04.4. The maximum time given to the algorithms for execution was 7200 seconds, i.e., 2 hours. Main memory usage was restricted to 32GB.

5.4.3 Results and Discussion

The 160 TYPE-1 benchmarks and 264 TYPE-2 benchmarks covered a wide spectrum in terms of the number of factors, the total number of variables and the number of variables to be eliminated. For instance, in the TYPE-1 category, the number of factors varied from 44 to 7034, total number of variables varied from 94 to 9782 and the number of variables to eliminate varied from 60 to 4751. Amongst the TYPE-2 benchmarks, the number of factors varied varied from 24 to 3956, the total number of variables varied from 70 to 5963, and the variables to eliminate varied 21 to 2689.

We first present *CegarSkolem* vs Bloqqer results and *CegarSkolem* vs DepQBF results on the TYPE-1 benchmarks followed by *CegarSkolem* vs *MonoSkolem* results on all the benchmarks. We then present the results of our experiments on the variants of *CegarSkolem* discussed in Subsection 5.3.5. We also discuss the results of experiments on *CegarSkolem* with different factor sizes, ordering, and generalization function *Generalize*.

***CegarSkolem* vs Bloqqer**

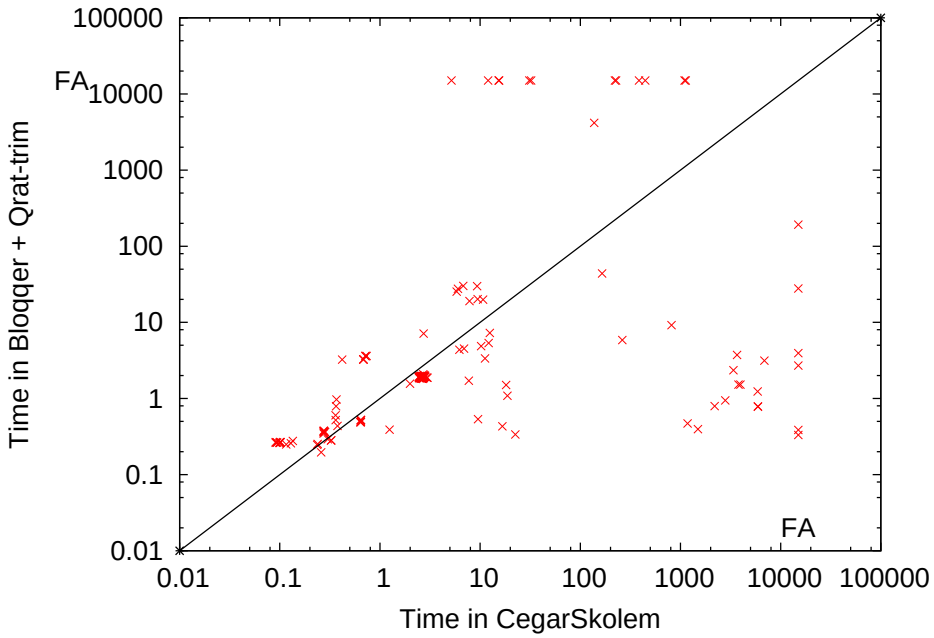

Figure 5.4: Time taken by *CegarSkolem* vs time taken by Bloqqer on TYPE-1 benchmarks (in seconds). Topmost points indicate benchmarks where Bloqqer couldn't generate Skolem functions; rightmost points indicate benchmarks where *CegarSkolem* couldn't generate Skolem functions.

Figure 5.4 shows the total time taken by *CegarSkolem* and Bloqqer on the 160 TYPE-1 benchmarks. Recall from Subsection 2.5.2 that bloqqer can generate Skolem functions only in the cases where validity of $\exists X.F(X,Y)$ can be established only by preprocessing. In other cases it gives a `NOT VERIFIED` message, and cannot generate Skolem functions. Among the 160 TYPE-1 benchmarks, Bloqqer could successfully generate Skolem functions for 148 benchmarks; it gave `NOT VERIFIED` message for the remaining 12. *CegarSkolem*, on the other hand, was able to successfully generate Skolem functions for 154 benchmarks.

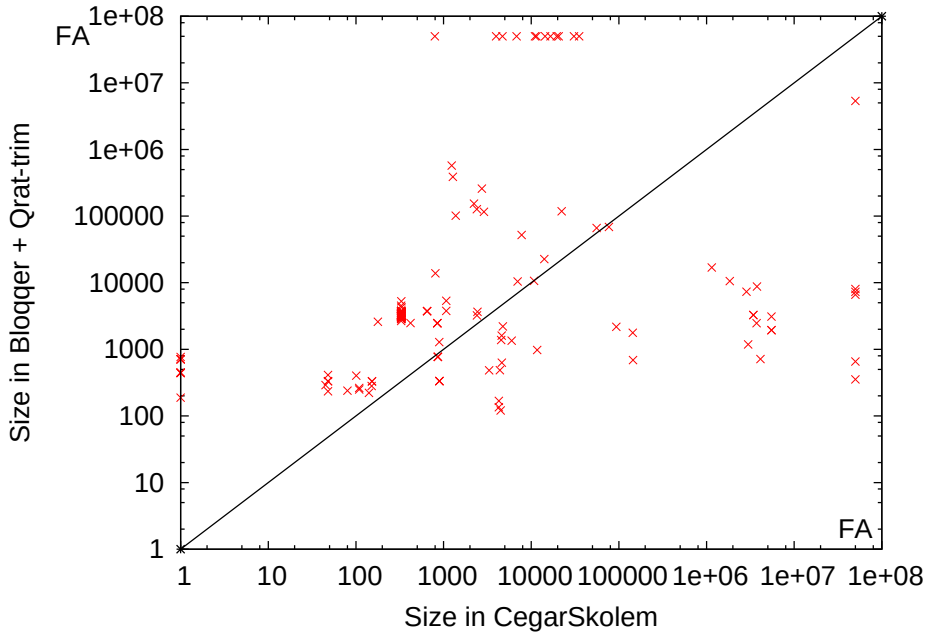

Figure 5.5: Maximum size of Skolem function by *CegarSkolem* vs maximum size of Skolem function by Bloqqer. Topmost points indicate benchmarks where Bloqqer couldn't generate Skolem functions; rightmost points indicate benchmarks where *CegarSkolem* couldn't generate Skolem functions.

In 4 cases *CegarSkolem* timed out, and in 2 cases it ran out of memory during construction of initial abstract Skolem functions in *InitAbsRef*.

Among the 142 remaining benchmarks which both *CegarSkolem* and Bloqqer successfully solved, for 88 benchmarks, time taken by both were comparable. The ratio of time taken by Bloqqer to time taken by *CegarSkolem* for these benchmarks was greater than 0.5 and less than 2.0. For 29 benchmarks *CegarSkolem* clearly outperformed Bloqqer. The ratio of time taken by Bloqqer to time taken by *CegarSkolem* for these benchmarks was 2.0 - 30.3. For 25 benchmarks, Bloqqer outperformed *CegarSkolem* by a factor of 2 or more. For 13 benchmarks among

them, the ratio of time taken by *CegarSkolem* to that by Bloqqer was 2.1 - 88.6. For 12 benchmarks, *CegarSkolem* took significantly more time compared to Bloqqer (almost 3 - 4 orders of magnitude). On profiling, we found that in all of these cases, more than 70% of the time taken by *CegarSkolem* was spent in the function *ReverseSubstitute*, due to larger sizes of Skolem functions generated.

There were 9 large benchmarks, with 1000+ factors and 1000+ variables to eliminate. Bloqqer could not generate Skolem functions for 8 of these and took $>$ 1 hour for the remaining one. *CegarSkolem* successfully generated Skolem functions for each of these in under 20 minutes.

For each Skolem function vector generated, we also compared in Figure 5.5 the maximum size of a Skolem function generated by the two algorithms. We chose to use the maximum size instead of average size for this comparison because of the following reason. A large number of auxiliary variables are introduced while converting a benchmark to `.qdimacs` format via Tseitin encoding. All such auxiliary variables must be existentially quantified, and the sizes of Skolem functions of these auxiliary variables are typically very small. This significantly skews down the average size of generated Skolem functions. Note that *CegarSkolem* does not require the benchmarks to be converted to `.qdimacs` format, and hence does not require Tseitin encoding, or introduction of auxiliary variables.

For most benchmarks, as shown in Figure 5.5, the maximum sizes of Skolem functions obtained by *CegarSkolem* were *smaller* than those generated by Bloqqer. Specifically, among the 142 benchmarks which both *CegarSkolem* and Bloqqer successfully solved, for 100 benchmarks, the ratio of maximum Skolem function size generated by Bloqqer to that by *CegarSkolem* was more than 2. For 89 of them the ratio was in the range 5.0 - 766.0. For 29 benchmarks Bloqqer generated

Skolem functions that are smaller than those generated by *CegarSkolem* by a factor of 2 or more. For 8 of them, the Skolem functions generated by *CegarSkolem* were bigger than those generated by Bloqqer by more than 3 orders of magnitude. Thus our analysis shows that CegarSkolem *performs better on larger benchmarks, and generates smaller Skolem functions on most benchmarks*.

***CegarSkolem* vs DepQBF**

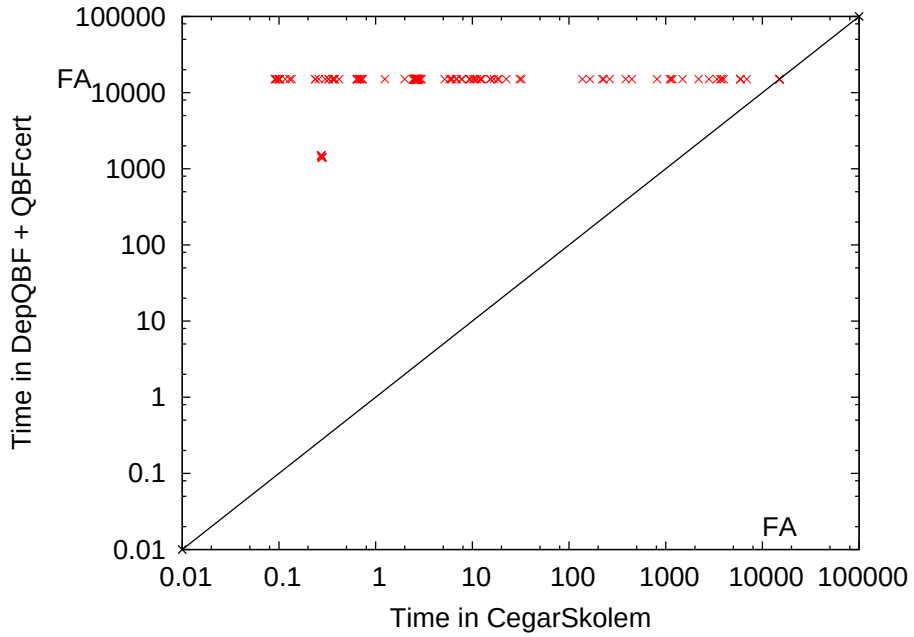

Figure 5.6: Time taken by *CegarSkolem* vs time taken by DepQBF on TYPE-1 benchmarks (in seconds). Topmost points indicate benchmarks where DepQBF couldn't generate Skolem functions; rightmost points indicate benchmarks where *CegarSkolem* couldn't generate Skolem functions.

Figure 5.6 shows the total time taken by *CegarSkolem* and DepQBF on the 160 TYPE-1 benchmarks. Figure 5.7 compares the maximum sizes of Skolem

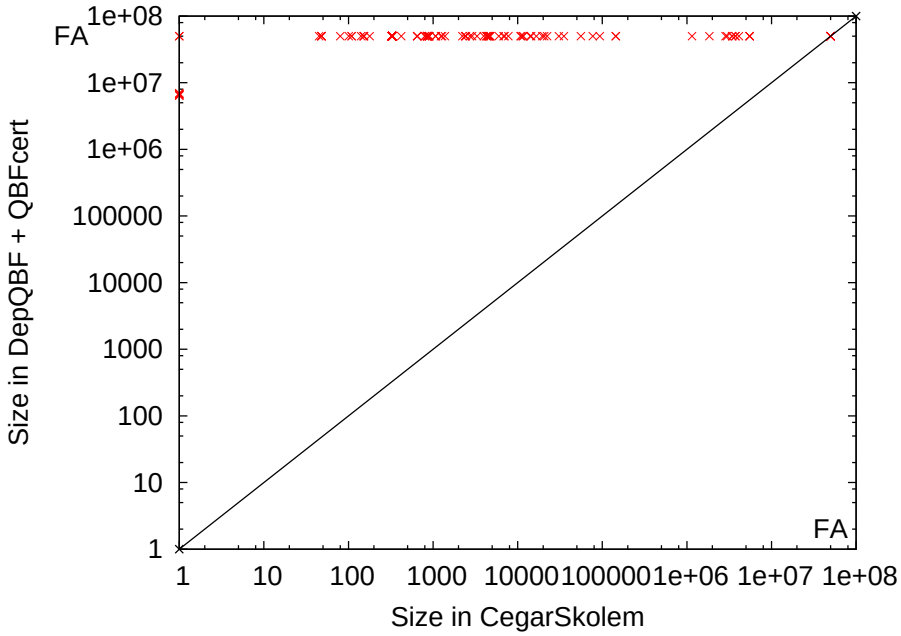

Figure 5.7: Maximum size of Skolem function by *CegarSkolem* vs maximum size of Skolem function by DepQBF. Topmost points indicate benchmarks where DepQBF couldn't generate Skolem functions; rightmost points indicate benchmarks where *CegarSkolem* couldn't generate Skolem functions.

functions generated by *CegarSkolem* with those generated by DepQBF. The results clearly demonstrate that *CegarSkolem outperforms DepQBF both in terms of time and Skolem function size on all benchmarks.*

Among the 160 TYPE-1 benchmarks, DepQBF could generate Skolem functions only for 7 benchmarks. For 150 benchmarks, the QRP proof files could not be generated; the files were huge ($>$ 32GB) and we ran out of disk space allocated. Among the remaining 10 benchmarks for which the QRP files could be generated, for 3 benchmarks, the QBFcert framework ran out of memory during extraction of Skolem functions from the QRP proofs.

CegarSkolem* vs *MonoSkolem

The performance of these two algorithms on both TYPE-1 and TYPE-2 categories of benchmarks is shown in Figure 5.8 and Figure 5.9. Figure 5.8 gives the average sizes of Skolem functions generated in a Skolem function vector. We measured average sizes here, since both algorithms generated Skolem functions for exactly the same set of variables. Figure 5.9 shows the total time taken in seconds. From Figure 5.8, it is clear that the Skolem functions generated by *CegarSkolem* are on an average *smaller* than those generated by *MonoSkolem*. *There is no instance on which* CegarSkolem *generates Skolem functions that are larger on average than those generated by* MonoSkolem.

Due to repeated calls to the SAT-solver, we expected *CegarSkolem* to take more time than *MonoSkolem*. From Figure 5.9, we can see that *CegarSkolem* does indeed take more time on some benchmarks but on most of them, the total time taken by both algorithms was *less than* 100 seconds. On profiling, we found that *CegarSkolem* spent most of its time on SAT solving. On 38 benchmarks where *CegarSkolem* took greater than 100 but less than 300 seconds, *MonoSkolem* performed significantly worse taking more than 1000 seconds. We found the degradation of *MonoSkolem* was due to the large sizes of Skolem functions generated; these were of the order of 1 million whereas those generated by *CegarSkolem* were less than 8000. *Large Skolem function sizes imply more time spent in function composition and reverse-substitution. This overhead was considerably greater than that incurred by SAT-solving, resulting in* MonoSkolem *taking significantly more time*.

We noticed that for 101 benchmarks where the sizes of Skolem functions generated were even larger (of the order of 10^7), *MonoSkolem* could not generate the

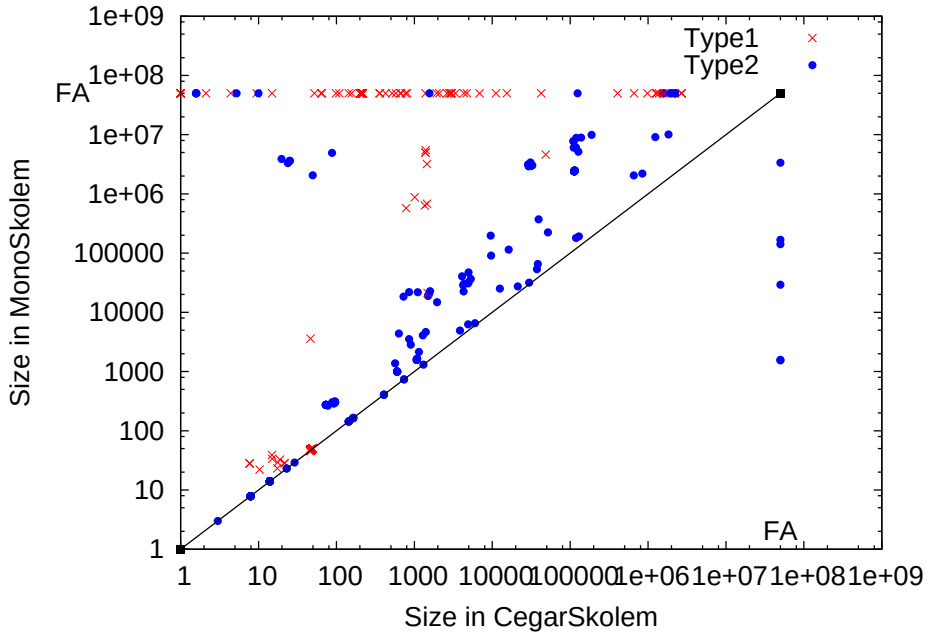

Figure 5.8: Average Skolem function size by *CegarSkolem* vs average Skolem function size by *MonoSkolem*. Topmost points indicate benchmarks where *MonoSkolem* couldn't generate Skolem functions; rightmost points indicate benchmarks where *CegarSkolem* couldn't generate Skolem functions.

Skolem functions. In 8 of these cases, the memory consumed by *MonoSkolem* increased rapidly, resulting in a memory out. In 10 cases, it ran out of time. In an overwhelming 83 cases, the Skolem functions generated during the execution of *MonoSkolem* were so huge that the integer used in the underlying ABC library for storing the level of AIG nodes overflowed. This resulted in assertion failure in the ABC library. Notice that *CegarSkolem* generated Skolem functions for all of these benchmarks. The rightmost points indicate 12 cases where *CegarSkolem* could not generate Skolem functions. Among these, 10 were time-outs and 2 were memory-outs. *MonoSkolem* succeeded in 6 of these cases. Both *CegarSkolem* and

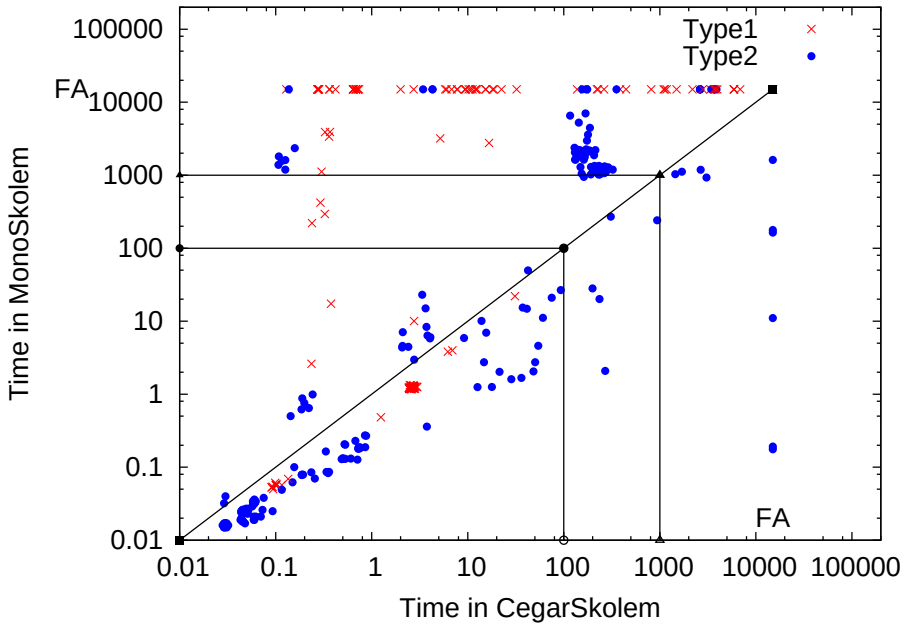

Figure 5.9: Time taken by *CegarSkolem* vs time taken by *MonoSkolem* (in seconds). Topmost points indicate benchmarks where *MonoSkolem* couldn't generate Skolem functions; rightmost points indicate benchmarks where *CegarSkolem* couldn't generate Skolem functions.

MonoSkolem failed on 6 cases; only Bloqqer could generate Skolem functions for these benchmarks. In the 2 memory-out cases for *CegarSkolem*, it ran out of memory during construction of initial abstract Skolem functions in *InitAbsRef*. *MonoSkolem* also ran out of memory in these cases.

Analysis of *CegarSkolem*

CegarSkolem could generate Skolem functions for 412 of the 424 benchmarks. For 197 of these benchmarks the initial abstract Skolem functions were correct, and most of the time was spent in the SAT solver. For the remaining 227 bench-

marks that *CegarSkolem* solved, the number of CEGAR iterations varied from 1 to 3121. For all benchmarks on which *CegarSkolem* timed out, we noticed that there were large subsets of factors that shared many variables in their supports. As a result, *CegarSkolem* could not exploit the factored representation effectively, requiring many refinements. On averaging over all benchmarks, we found that more than 33% of the time spent by *CegarSkolem* was for SAT-solving. This leads to the natural suggestion that we could use more efficient SAT solvers to improve the performance of *CegarSkolem*. Since SAT-solving technology continues to improve every year, we hope to leverage this to improve the performance of *CegarSkolem* further.

Variants of *CegarSkolem*

When optimizations using interpolants were enabled, the performance of *CegarSkolem* crucially depended on the size of the interpolants computed and time to compute the interpolants. However, we observed that the interpolants generated by ABC were often not succinct, and computing the interpolants was often time-intensive.

We measured the time taken by *CegarSkolem_Mod1* and *CegarSkolem_Mod2* for generation of Skolem functions for the benchmarks, and the average sizes of the Skolem functions generated. We compared the average sizes of the Skolem functions generated by *CegarSkolem_Mod1* with those generated by *CegarSkolem* (see Fig 5.10). The comparison clearly indicated that the Skolem functions generated by *CegarSkolem* were smaller in size compared to those generated by *CegarSkolem_Mod1*. Recall that the Skolem functions generated by *CegarSkolem* are of the form $\neg \mathtt{r1}[i]$, whereas the Skolem functions generated by *CegarSkolem_Mod1*

have the form $\neg \mathtt{r1}[i] \vee \mathtt{r0}[i]_{init}$. Bigger Skolem functions in *CegarSkolem_Mod1* were due to the additional disjunct $\mathtt{r0}[i]_{init}$ in the Skolem functions.

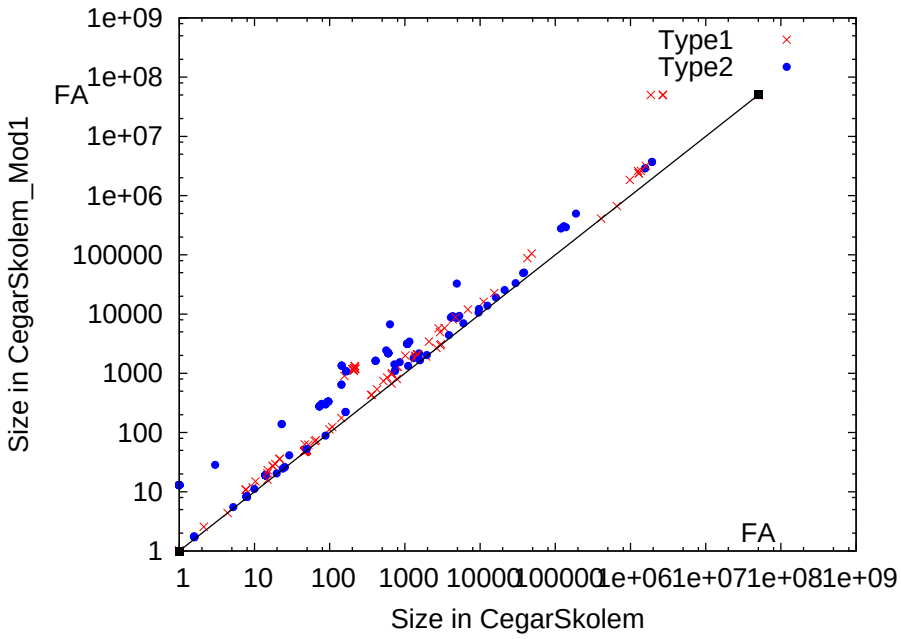

Figure 5.10: Average Skolem function size by *CegarSkolem* vs average Skolem function size by *CegarSkolem_Mod1*. Topmost points indicate benchmarks where *CegarSkolem_Mod1* couldn't generate Skolem functions; rightmost points indicate benchmarks where *CegarSkolem* couldn't generate Skolem functions.

We then compared (in Fig 5.11) the time taken by *CegarSkolem_Mod1* for generation of Skolem functions with the time taken by *CegarSkolem*. We also compared the number of CEGAR iterations needed by *CegarSkolem_Mod1* with the number of CEGAR iterations needed by *CegarSkolem* for the benchmarks where both the algorithms succeeded in generating Skolem functions (see Fig 5.12) (iterations+1 used in the plot to include cases with zero iterations).

The time taken by both the algorithms were roughly the same for almost

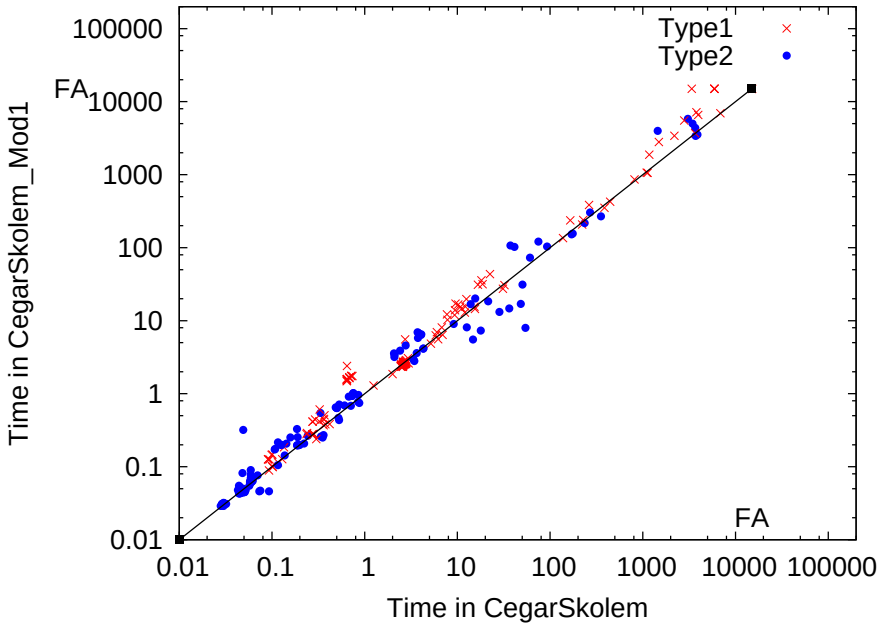

Figure 5.11: Time taken by *CegarSkolem* vs time taken by *CegarSkolem_Mod1* (in seconds). Topmost points indicate benchmarks where *CegarSkolem_Mod1* couldn't generate Skolem functions; rightmost points indicate benchmarks where *CegarSkolem* couldn't generate Skolem functions.

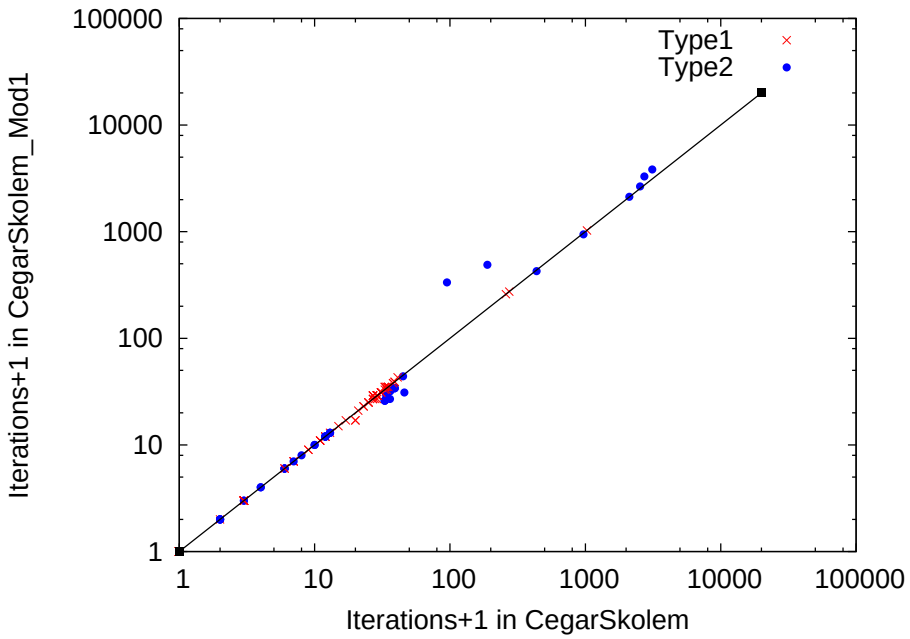

Figure 5.12: Number of CEGAR iterations by *CegarSkolem* vs number of CEGAR iterations by *CegarSkolem_Mod1*.

all the benchmarks. We observed that due to bigger sizes of Skolem functions, *CegarSkolem_Mod1* spent more time in reverse-substitution compared to that spent by *CegarSkolem*. Recall that the error formula ε' in *CegarSkolem_Mod1* differs from the error formula ε in *CegarSkolem* in two ways: (i) the abstract Skolem function ψ_i^A is $\neg \mathtt{r1}[i] \vee \mathtt{r0}[i]_{init}$ instead of $\neg \mathtt{r1}[i]$, and (ii) $(\text{bad}_1 \vee \ldots \vee \text{bad}_{n-1})$ is used in place of $\neg F(X,Y)$. For each benchmark, we measured the total time taken by the SAT solver calls that check the satisfiability of ε in *CegarSkolem*. We call this ε-time for the benchmark. Similarly for each benchmark we measured the total time taken by the SAT solver calls that check the satisfiability of ε' in *CegarSkolem_Mod1*. We call this ε'-time for the benchmark. It was observed that ε'-time is greater than ε-time for many benchmarks. There were benchmarks with ε'-time $<$ ε-time, but in most of these cases, the time saved in SAT calls was less than the additional time incurred in reverse-substitution.

There were 4 benchmarks on which *CegarSkolem_Mod1* timed out and *CegarSkolem* could generate the Skolem functions. Interestingly, on these benchmarks *CegarSkolem* performed worse when compared to Bloqqer due to huge Skolem functions and consequent expensive reverse-substitution. The Skolem functions generated inside *CegarSkolem_Mod1* for these benchmarks were even bigger which resulted in timing out inside the reverse-substitution phase.

We compared the average sizes of Skolem functions generated by *CegarSkolem_Mod2* with those generated by *CegarSkolem* (see Fig 5.13), and the time taken by *CegarSkolem_Mod2* for generation of Skolem functions with the time taken by *CegarSkolem* (see Fig 5.15). We also compared the number of CEGAR iterations needed by *CegarSkolem_Mod2* with the number of CEGAR iterations needed by *CegarSkolem* (see Fig 5.14).Our analysis indicated that both the algorithms be-

haved very similar in all the cases, resulting in similar execution times and Skolem function sizes. Considerable differences in execution times were observed only in a *very few* cases. We found that these differences were due to differences in the time spent in SAT calls.

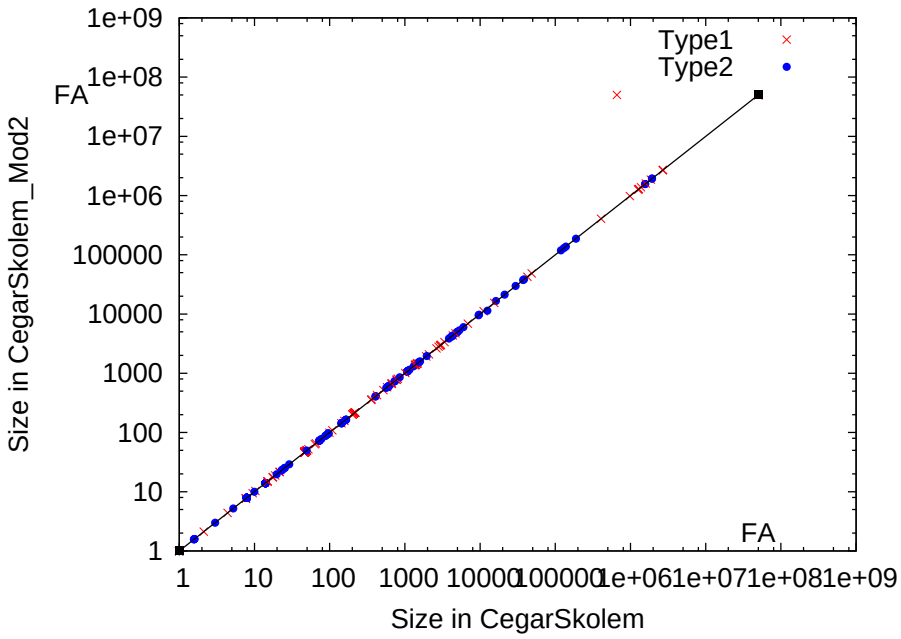

Figure 5.13: Average Skolem function size by *CegarSkolem* vs average Skolem function size by *CegarSkolem_Mod2*. Topmost points indicate benchmarks where *CegarSkolem_Mod2* couldn't generate Skolem functions; rightmost points indicate benchmarks where *CegarSkolem* couldn't generate Skolem functions.

Effect of ordering on *CegarSkolem*

In the variable ordering that we used in *CegarSkolem*, variables occurring in fewer factors are indexed lower than those occurring in more factors. In order to understand the effect of this ordering on the performance of *CegarSkolem*, we used a

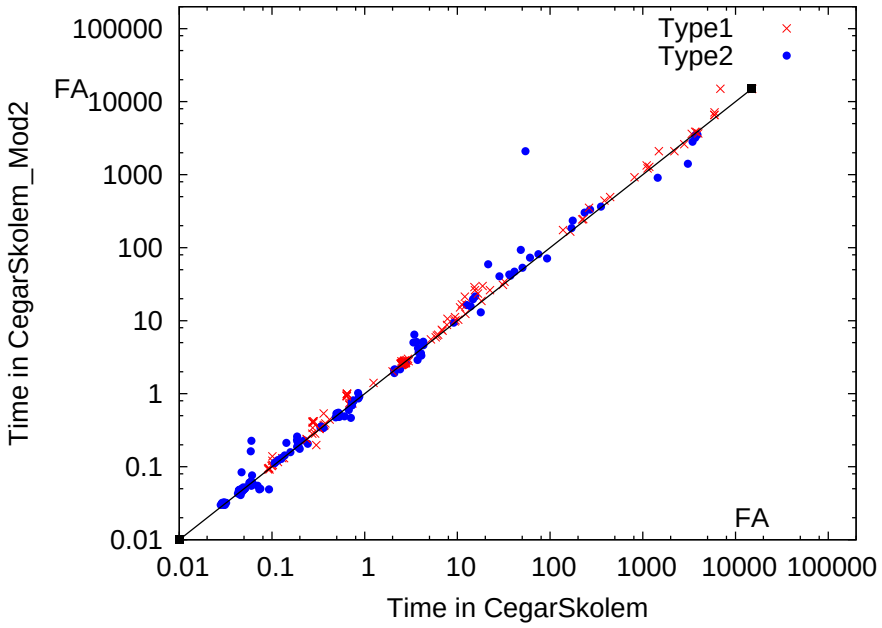

Figure 5.14: Time taken by *CegarSkolem* vs time taken by *CegarSkolem_Mod2* (in seconds). Topmost points indicate benchmarks where *CegarSkolem_Mod2* couldn't generate Skolem functions; rightmost points indicate benchmarks where *CegarSkolem* couldn't generate Skolem functions.

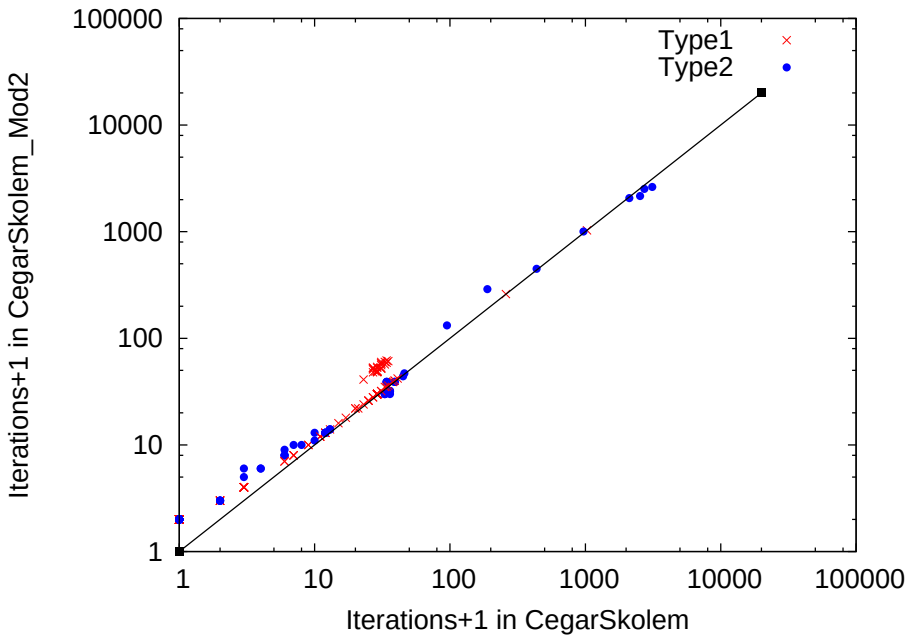

Figure 5.15: Number of CEGAR iterations by *CegarSkolem* vs number of CEGAR iterations by *CegarSkolem_Mod2*.

variant of *CegarSkolem* called *CegarSkolem_Lexico* that uses lexicographical ordering on the variables.

We measured the time taken by *CegarSkolem_Lexico* for generation of Skolem functions for the benchmarks, and the average sizes of the Skolem functions generated. In Fig 5.16 we compare the average sizes of the Skolem functions generated by *CegarSkolem_Lexico* with those generated by *CegarSkolem*, and in Fig 5.17 we compare the time taken by *CegarSkolem_Lexico* with the time taken by *CegarSkolem*.

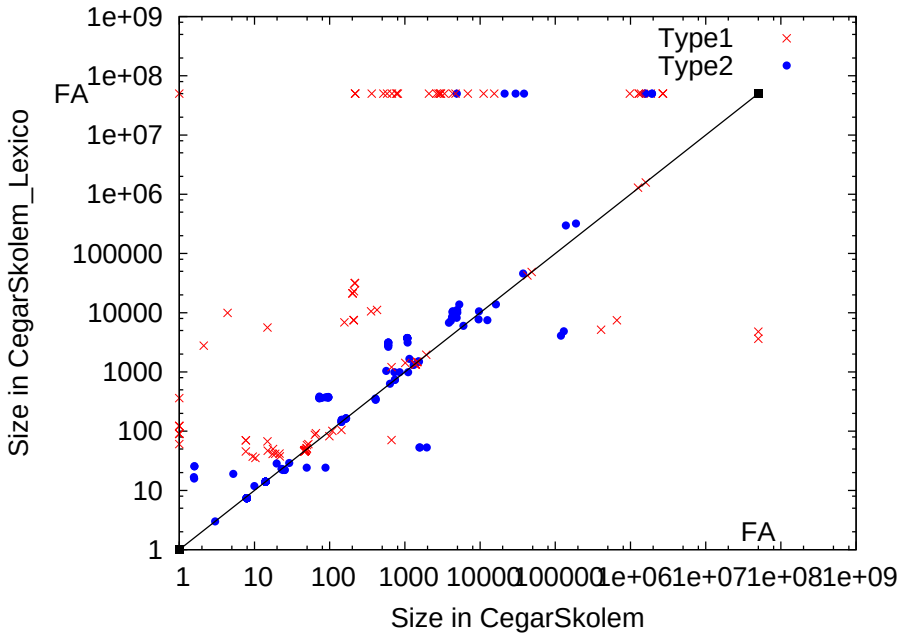

Figure 5.16: Average Skolem function size by *CegarSkolem* vs average Skolem function size by *CegarSkolem_Lexico*. Topmost points indicate benchmarks where *CegarSkolem_Lexico* couldn't generate Skolem functions; rightmost points indicate benchmarks where *CegarSkolem* couldn't generate Skolem functions.

The results clearly indicate that the ordering that we used in *CegarSkolem*

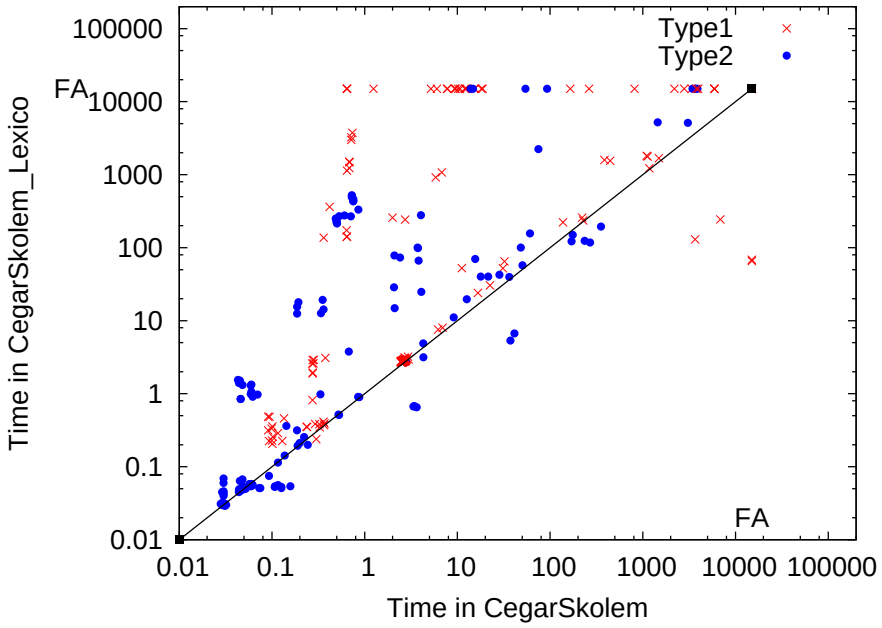

Figure 5.17: Time taken by *CegarSkolem* vs time taken by *CegarSkolem_Lexico* (in seconds). Topmost points indicate benchmarks where *CegarSkolem_Lexico* couldn't generate Skolem functions; rightmost points indicate benchmarks where *CegarSkolem* couldn't generate Skolem functions.

gives better performance. In order to analyze the results in detail, we compared the number of CEGAR iterations needed by *CegarSkolem_Lexico* with the number of CEGAR iterations needed by *CegarSkolem* (see Fig 5.18).

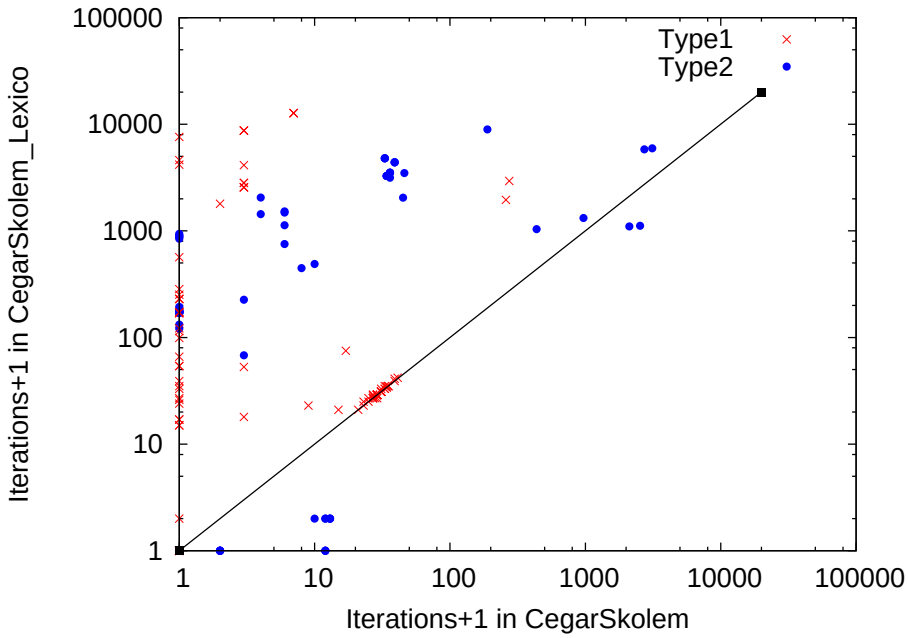

Figure 5.18: Number of CEGAR iterations by *CegarSkolem* vs number of CEGAR iterations by *CegarSkolem_Lexico*.

Our analysis revealed that, in general, the abstract Skolem functions generated by *InitAbsRef* in *CegarSkolem* were more refined compared to those generated by *InitAbsRef* in *CegarSkolem_Lexico*. Due to this, *CegarSkolem_Lexico* spent more time inside the CEGAR loop compared to *CegarSkolem*. This also resulted in bigger Skolem functions in *CegarSkolem_Lexico*, and consequently more time in reverse-substitution.

Effect of choice of factors on *CegarSkolem*

In order to understand how a different choice of factors would affect the performance of *CegarSkolem*, we used a variant of *CegarSkolem* called *CegarSkolem_Clause*. *CegarSkolem_Clause* takes benchmarks in .qdimacs format as input. In the .qdimacs version of a benchmark $\exists X.F(X,Y)$, the formula $F(X,Y)$ appears as a conjunction of clauses, and thus each clause becomes a factor.

We executed *CegarSkolem_Clause* on the .qdimacs versions of TYPE-1 benchmarks, and measured the time taken for generation of Skolem functions and the maximum sizes of the Skolem functions generated. We chose to use the maximum sizes of Skolem functions instead of average sizes, since *CegarSkolem_Clause* takes benchmarks in .qdimacs format as input. As mentioned before, the sizes of Skolem functions of the Tseitin variables in the .qdimacs format are typically very small, which significantly skews down the average Skolem function sizes. In Fig 5.19 we compare the maximum sizes of the Skolem functions generated by *CegarSkolem_Clause* with those generated by *CegarSkolem*, and in Fig 5.20 we compare the time taken by *CegarSkolem_Clause* with the time taken by *CegarSkolem*. We also measured the number of CEGAR iterations needed by *CegarSkolem_Clause*, and compared with the number of CEGAR iterations needed by *CegarSkolem* (see Fig 5.21).

The initial abstract Skolem functions generated by *InitAbsRef* inside *CegarSkolem_Clause* were significantly smaller than those generated inside *CegarSkolem*. However the initial abstract Skolem functions generated inside *CegarSkolem_Clause* were more abstract compared to those generated inside *CegarSkolem*. As a result of this, as shown in Fig 5.21, the number of CEGAR iterations needed by *CegarSkolem_Clause* were significantly more than the number of CEGAR iterations

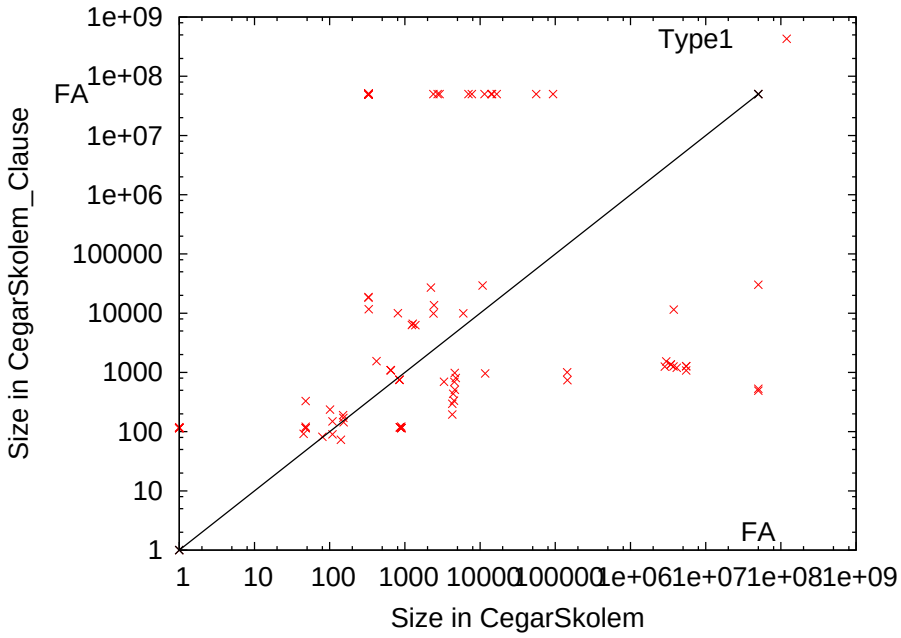

Figure 5.19: Maximum Skolem function size by *CegarSkolem* vs maximum Skolem function size by *CegarSkolem_Clause*. Topmost points indicate benchmarks where *CegarSkolem_Clause* couldn't generate Skolem functions; rightmost points indicate benchmarks where *CegarSkolem* couldn't generate Skolem functions.

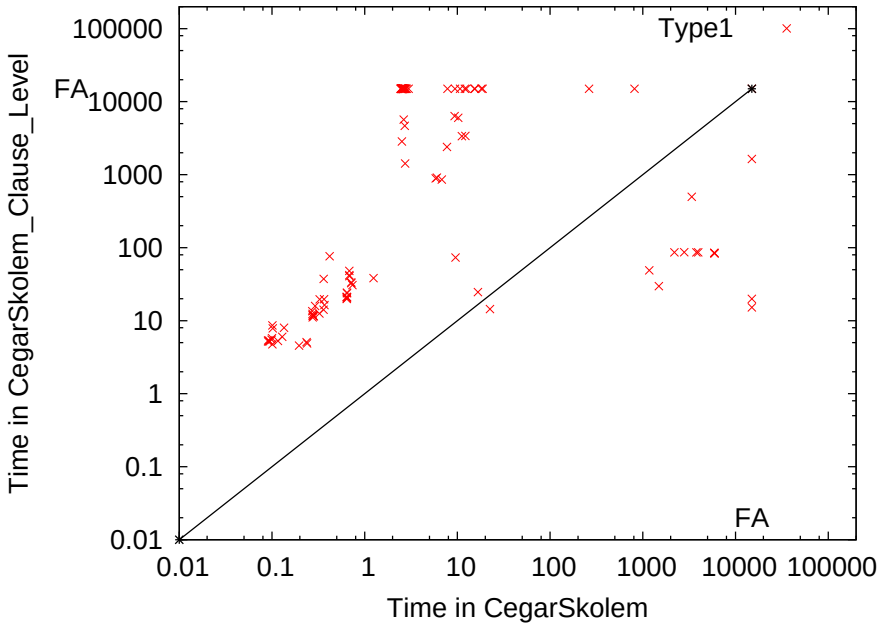

Figure 5.20: Time taken by *CegarSkolem* vs time taken by *CegarSkolem_Clause* (in seconds). Topmost points indicate benchmarks where *CegarSkolem_Clause* couldn't generate Skolem functions; rightmost points indicate benchmarks where *CegarSkolem* couldn't generate Skolem functions.

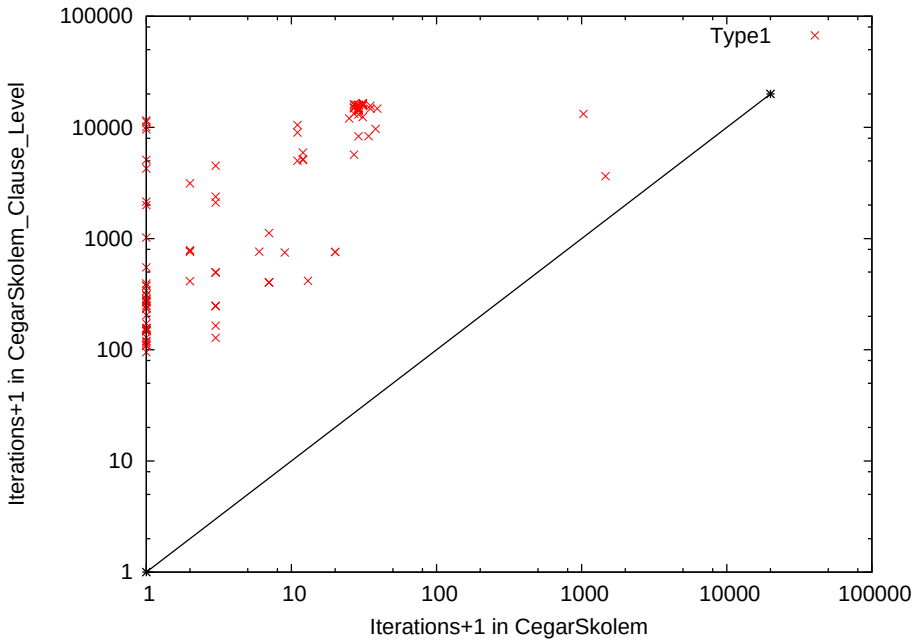

Figure 5.21: Number of CEGAR iterations by *CegarSkolem* vs number of CEGAR iterations by *CegarSkolem_Clause*.

needed by *CegarSkolem*.

In most of the benchmarks, due to the increased number of CEGAR iterations, *CegarSkolem_Clause* performed worse compared to *CegarSkolem*. However, the smaller initial abstract Skolem functions helped *CegarSkolem_Clause* in 13 benchmarks, where it clearly outperformed *CegarSkolem* (see Fig 5.20). In 11 of these cases, *CegarSkolem* performed bad due to huge Skolem functions and consequent expensive reverse-substitution. In 2 cases *CegarSkolem* ran out of memory during construction of initial abstract Skolem functions in *InitAbsRef*. Although the number of CEGAR iterations in *CegarSkolem_Clause* were much more than those in *CegarSkolem* for these benchmarks, the simpler initial Skolem functions helped *CegarSkolem_Clause* to avoid the blow-up in Skolem function sizes.

Experiments with different *Generalize*

In *CegarSkolem* we used the function *Generalize* that takes as arguments, an assignment π and a function φ such that $\pi \models \varphi$, and returns a function ξ that generalizes π while satisfying φ. Recall that φ here is a disjunction of functions which can also be viewed as a set of functions. Our implementation of *Generalize*(π, φ) in *CegarSkolem* returns *one of the functions* in φ (viewed as a set) that evaluates to 1 under π. There are several other ways of implementing *Generalize*(π, φ). For example, *Generalize*(π, φ) can return the disjunction of *all functions* in φ (viewed as a set) that evaluate to 1 under π. We call the variant of *CegarSkolem* that uses this implementation of *Generalize*(π, φ) as *CegarSkolem_Mod3*.

We measured the time taken by *CegarSkolem_Mod3* for generation of Skolem functions for the benchmarks, and the average sizes of the Skolem functions

generated. In Fig 5.22 we compare the average sizes of the Skolem functions generated by *CegarSkolem_Mod3* with those generated by *CegarSkolem*, and in Fig 5.23 we compare the time taken by *CegarSkolem_Mod3* with the time taken by *CegarSkolem*.

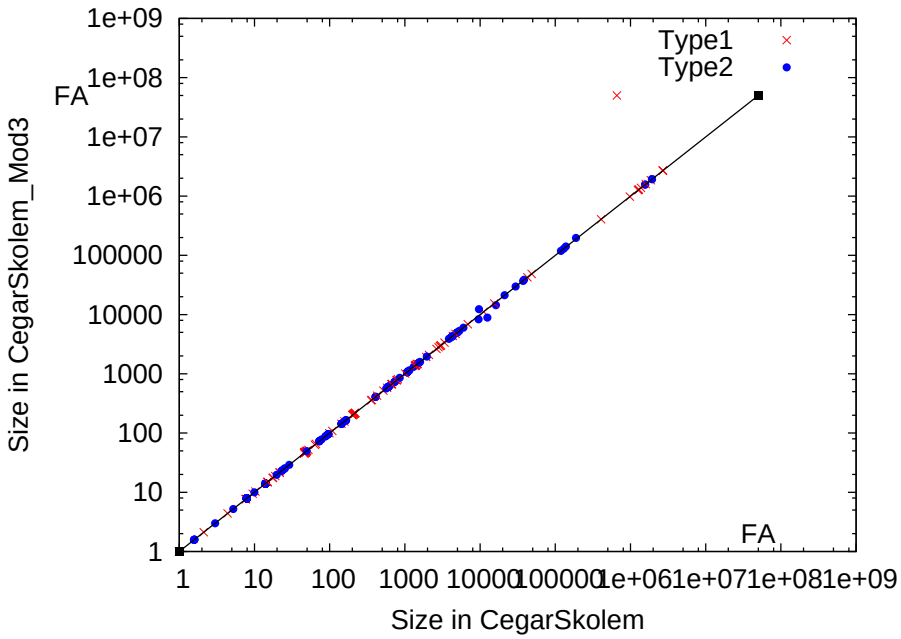

Figure 5.22: Average Skolem function size by *CegarSkolem* vs average Skolem function size by *CegarSkolem_Mod3*. Topmost points indicate benchmarks where *CegarSkolem_Mod3* couldn't generate Skolem functions; rightmost points indicate benchmarks where *CegarSkolem* couldn't generate Skolem functions.

We observed that both the algorithms behaved very similar on all benchmarks, resulting in similar execution times and Skolem function sizes. There were many cases where the number of CEGAR iterations needed by *CegarSkolem_Mod3* was less than those needed by *CegarSkolem* (see Fig 5.24). However, the functions returned by *Generalize*(π, φ) calls in *CegarSkolem_Mod3* were bigger in size when

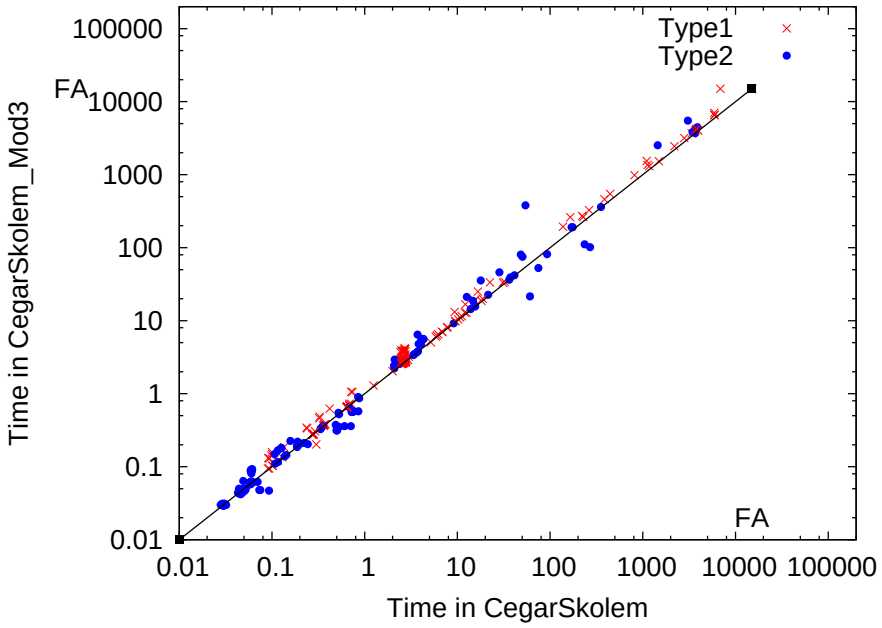

Figure 5.23: Time taken by *CegarSkolem* vs time taken by *CegarSkolem_Mod3* (in seconds). Topmost points indicate benchmarks where *CegarSkolem_Mod3* couldn't generate Skolem functions; rightmost points indicate benchmarks where *CegarSkolem* couldn't generate Skolem functions.

compared to those returned by *Generalize*(π, φ) calls in *CegarSkolem*. Hence although the number of CEGAR iterations, and consequently number of invocations of *UpdateAbsRef* and SAT checks in *CegarSkolem_Mod3* were fewer than those in *CegarSkolem*, the individual invocations of *UpdateAbsRef* and SAT checks in *CegarSkolem_Mod3* were more expensive. Effectively the execution times of the algorithms did not show much difference.

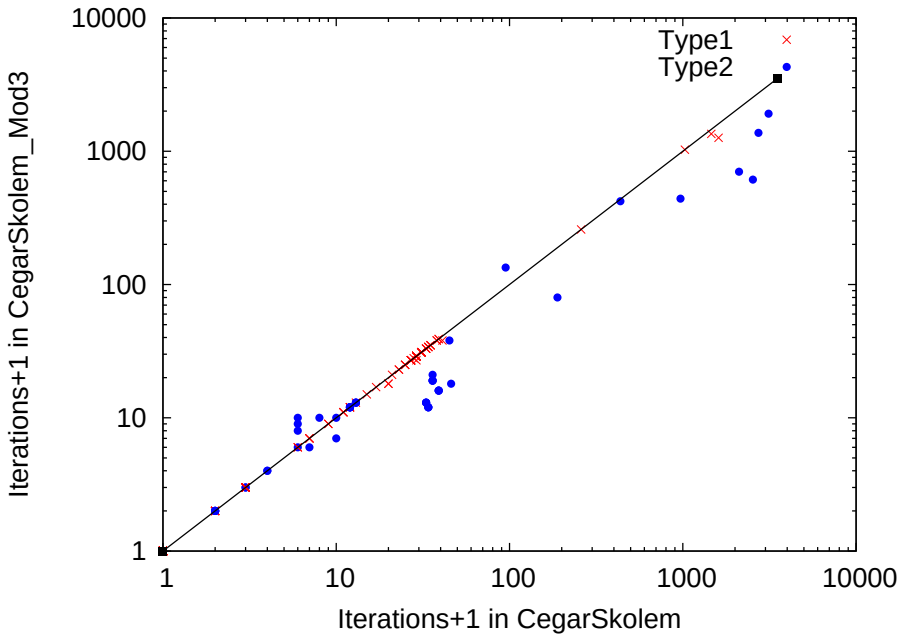

Figure 5.24: Number of CEGAR iterations by *CegarSkolem* vs number of CEGAR iterations by *CegarSkolem_Mod3*.

5.5 Conclusions

We presented a CEGAR algorithm for generating Skolem functions from factored propositional formulas. Our experiments show that for complex functions, our

algorithm significantly out-performs state-of-the-art algorithms.

Our techniques for generation of Skolem functions can be extended to more generic cases. One of these is the case where the formula $F(X,Y)$ involves uninterpreted predicates. Formally, given a Boolean formula $F(X,Y)$ on variables in $X \cup Y$ involving uninterpreted predicates $P_1,\ldots,P_n$, we wish to generate Skolem function $\psi(X \setminus x_i, Y)$ for $x_i \in X$ in $F(X,Y)$. It can be observed that the results that we proved in this chapter hold in spite of the presence of uninterpreted predicates. As an example, Let X be (x_1,x_2) and Y be (y_1). Let $F(X,Y)$ be the formula $P_1(x_2) \wedge x_1 \wedge y$, where P_1 is an uninterpreted predicate. Suppose we wish to compute Skolem function vector (ψ_1,ψ_2) for (x_1,x_2) in F. Note that ψ_1 can be computed as $F[x_1 \mapsto 1] = P_1(x_2) \wedge y$. Now $\exists x_1.F$ is $F[x_1 \mapsto \psi_1]$, i.e., $P_1(x_2) \wedge y$. In a similar manner, ψ_2 can be computed as $P_1(1) \wedge y$. After substituting ψ_2 for x_2 in ψ_1, we have $(\psi_1,\psi_2) = (P_1(P_1(1) \wedge y) \wedge y,\ P_1(1) \wedge y)$.

Techniques for generating Skolem functions have many other potential applications which we have not fully explored. For example, suppose we have a Boolean formula $F(X,Y)$ on variables in $X \cup Y$, where X is $\{x_1,\ldots,x_n\}$. Moreover, suppose $F(x_1',x_2,\ldots,x_n,Y) \wedge F(x_1'',x_2,\ldots,x_n,Y) \wedge (x_1' \neq x_1'')$ is unsatisfiable, where x_1' and x_1'' are fresh variables. In this case, the value of x_1 is completely determined by the values of variables in $X \setminus x_1 \cup Y$. We call x_1 a dependent variable and the variables in $X \setminus x_1 \cup Y$ independent variables. The problem of expressing x_1 as a function of the independent variables reduces to finding Skolem function $\psi(X \setminus x_1, Y)$ for x_1 in F. We would like to explore such applications of Skolem functions in future.

Chapter 6

Conclusions and Future Works

This dissertation presented techniques to improve the scalability of formal verification and analysis tools for hardware and software systems.

We presented practically efficient and bit-precise techniques for quantifier elimination from linear modular constraints. Our techniques outperform alternate quantifier elimination techniques, and keep the final result in modular arithmetic. Many key operations performed by underlying algorithms in formal verification and analysis tools essentially boil down to quantifier elimination from formulas involving linear modular constraints. Specifically we showed the utility of our techniques in one such formal verification activity — in computing abstract symbolic transition relations for improving the scalability of bounded model checking of word-level RTL designs.

We also presented an efficient algorithm to generate succinct Skolem functions for propositional formulas. Our algorithm exploits the factored form of input formulas, and directly benefits from advances in SAT solving technology. Moreover unlike existing techniques in literature, our algorithm neither requires proof of

satisfiability nor resorts to memory-intensive compositions. The algorithm finds application in disjunctive decomposition of sequential circuits which is useful in improving the scalability of reachability analysis.

There are several promising directions for future work.

– Developing a quantifier elimination procedure for full bit-vector arithmetic is an interesting research direction. Other than linear modular arithmetic operations, bit-vector arithmetic primarily includes extractions, concatenations, non-linear multiplications and bit-wise operations [13]. The work by Bruttomesso et. al. in [110] shows that constraints in the core bit-vector theory consisting of only equalities, extractions and concatenations can be equivalently expressed as a conjunction of equalities on slices of variables involved in the constraints. The slices can be replaced by fresh variables to generate equisatisfiable linear modular equalities. It is interesting to see how our quantifier elimination techniques can be extended to handle constraints involving non-linear multiplications. Bit-wise operations may not be amenable to word-level reasoning and may require bit-level quantifier elimination. A layered framework like that of ours looks promising for such a quantifier elimination procedure for the full bit-vector arithmetic, since the amenability to word-level reasoning and simplifications varies considerably across different operations in the bit-vector theory.

– Quantifier elimination problem instances that arise in practice frequently mix expressions from different theories. The problem of reasoning on formulas in combined theories is well studied in the context of decision procedures. There are well established techniques such as Nelson-Oppen combination method [111] that helps to come up with a decision procedure for

a combined theory, given decision procedures for the individual theories. However the problem of *quantifier elimination from combined theories* is largely unaddressed to the best of our knowledge. Specifically it is interesting to study how the quantifier elimination techniques introduced in this thesis can be extended to work in combined theories such as combination of linear modular arithmetic and equality over uninterpreted functions, combination of linear modular arithmetic and array logic etc.

– Another interesting research direction is to understand how the ideas introduced in this dissertation can be used in decision procedures for bit-vector formulas. Clearly some of the techniques we proposed such as identifying and dropping unconstraining LMIs and LMDs can be useful in bit-vector solvers. Given a conjunction of LMCs, dropping unconstraining LMIs and LMDs gives another conjunction of LMCs which is potentially simpler and equisatisfiable to the original conjunction of LMCs. This can be beneficial in performing word-level simplifications in lazy and layered bit-vector solving frameworks such as that of MathSAT [84]. Recall from Section 4.4 that our limited preliminary experiments in this direction gave mixed results.

– Our CEGAR based algorithm for generation of Skolem functions admits optimizations which we would like to explore as part of future work. One such important optimization is the opportunity to refine using multiple counterexamples in parallel. This would allow us to significantly improve on our run times. Building portfolio Skolem function generators that run several algorithms in parallel is yet another direction to explore.

Bibliography

[1] E. M. Clarke, O. Grumberg, D. Peled. *Model checking*, MIT Press, 1999.

[2] D. Kapur. *A quantifier-elimination based heuristic for automatically generating inductive assertions for programs*, In Journal of Systems Science and Complexity, Volume 19, Issue 3, Pages 307-330, 2006.

[3] S. Das. *Predicate abstraction*, PhD dissertation, Stanford University, 2003.

[4] A. Bierre, A. Cimatti, E. M. Clarke, Y. Zhu. *Symbolic model checking without BDDs*, In Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Pages 193-207, 1999.

[5] H. Enderton. *A mathematical introduction to logic*, Academic Press, 2001.

[6] A. Trivedi. *Techniques in symbolic model checking*, Master's thesis, Indian Institute of Technology Bombay, Mumbai, India, 2003.

[7] D. Thomas, S. Chakraborty, P. Pandya. *Efficient guided symbolic reachability using reachability expressions*, In Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Pages 120-134, 2006.

[8] M. Benedetti. *Extracting certificates from quantified boolean formulas*, In Proceedings of International Joint Conferences on Artificial Intelligence (IJCAI), Pages 47-53, 2005.

[9] S. Srivastava, S. Gulwani, J. Foster. *Template-based program verification and program synthesis*, In International Journal on Software Tools for Technology Transfer (STTT), Volume 15, Issue 5, Pages 497-518, 2013.

[10] L. de Moura, N. Bjørner. *Z3: An efficient SMT solver*, In Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Pages 337-340, 2008.

[11] W. Pugh. *The Omega Test: A fast and practical integer programming algorithm for dependence analysis*, In Communications of the ACM, Volume 35, Issue 8, Pages 102-114, 1992.

[12] N. Bjørner, A. Blass, Y. Gurevich, M. Musuvathi. *Modular difference logic is hard*, In CoRR abs/0811.0987, 2008.

[13] D. Kroening, O. Strichman. *Decision procedures: An algorithmic point of view*, Springer, 2008.

[14] CUDD release 2.4.2 website, `vlsi.colorado.edu/~fabio/CUDD`.

[15] V. Ganesh, S. Berezin, D. Dill. *Deciding Presburger arithmetic by model checking and comparisons with other methods*, In Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), Pages 171-186, 2002.

[16] J.-H. R. Jiang, V. Balabanov. *Resolution proofs and Skolem functions in QBF evaluation and applications*, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 149-164, 2011.

[17] M. J. H. Heule, M. Seidl, A. Biere. *Efficient extraction of Skolem functions from QRAT proofs*, In Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), Pages 107-114, 2014.

[18] J.-H. R. Jiang. *Quantifier elimination via functional composition*, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 383-397, 2009.

[19] W. Craig. *Linear reasoning: A new form of the Herbrand-Gentzen theorem*, In Journal of Symbolic Logic, Volume 22, Issue 3, Pages 250-268, 1957.

[20] A. John, S. Chakraborty. *A quantifier elimination algorithm for linear modular equations and disequations*, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 486-503, 2011.

[21] A. John, S. Chakraborty. *Extending quantifier elimination to linear inequalities on bit-vectors*, In Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Pages 78-92, 2013.

[22] A. John, S. Chakraborty. *A layered algorithm for quantifier elimination from linear modular constraints*, In Formal Methods in System Design, Volume 49, Issue 3, Pages 272-323, 2016.

[23] A. John, S. Shah, S. Chakraborty, A. Trivedi, S. Akshay. *A CEGAR algorithm for generating Skolem functions for factored formulas*, In Proceedings

of International Conference on Formal Methods in Computer-Aided Design (FMCAD), Pages 85-92, 2015.

[24] K. L. McMillan. *Applying SAT methods in unbounded symbolic model checking*, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 250-264, 2002.

[25] J. Brauer, A. King, J. Kriener. *Existential quantification as incremental sat*, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 191-207, 2011.

[26] E. Goldberg, P. Manolios. *Quantifier elimination via clause redundancy*, In Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), Pages 85-92, 2013.

[27] R. Cavada, A. Cimatti, A. Franzen, K. Kalyanasundaram, M. Roveri, R. K. Shyamasundar. *Computing predicate abstractions by integrating BDDs and SMT solvers*, In Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), Pages 69-76, 2007.

[28] S. Chaki, A. Gurfinkel, O. Strichman. *Decision diagrams for linear arithmetic*, In Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), Pages 53-60, 2009.

[29] D. Monniaux. *A quantifier elimination algorithm for linear real arithmetic*, In Proceedings of International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR), Pages 243-257, 2008.

[30] A. Bradley, Z. Manna. *Calculus of computation: Decision procedures with applications to verification*, Springer, 2007.

[31] J. Ferrante, C. Rackoff. *A decision procedure for the first order theory of real addition with order*, In Society for Industrial and Applied Mathematics (SIAM) Journal on Computing, Volume 4, Issue 1, Pages 69-76, 1975.

[32] R. Loos, V. Weispfenning. *Applying linear quantifier elimination*, In Computer Journal, Volume 36, Issue 5, Pages 450-462, 1993.

[33] B. Boigelot, S. Jodogne, P. Wolper. *On the use of weak automata for deciding linear arithmetic with integer and real variables*, In Proceedings of International Joint Conference on Automated Reasoning (IJCAR), Pages 611-625, 2001.

[34] B. Becker, C. Dax, J. Eisinger, F. Klaedtke. *LIRA: Handling constraints of linear arithmetics over the integers and the reals*, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 307-310, 2007.

[35] The LASH toolset. `http://www.montefiore.ulg.ac.be/boigelot/research/lash/`.

[36] L. Khachian. *A polynomial time algorithm in linear programming*, In Doklady Akademii Nauk SSSR, Volume 244, Pages 1093-1096, 1979.

[37] N. Karmarkar. *A new polynomial-time algorithm for linear programming*, In Combinatorica, Volume 4, Issue 4, Pages 373-397, 1984.

[38] B. Dutertre, L. de Moura. *A fast linear-arithmetic solver for DPLL(T)*, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 81-94, 2006.

[39] G. Dantzig. *Linear programming and extensions*, Princeton University Press, 1963.

[40] R. Stansifer. *Presburger's article on integer arithmetic: Remarks and translation*, Technical Report, Computer Science Department, Cornell University, 1984.

[41] D. Cooper. *Theorem proving in arithmetic without multiplication*, In Machine Intelligence, Volume 7, Pages 91-99, 1972.

[42] J. Büchi. *Weak second-order arithmetic and finite automata*, In Zeitschrift der mathematischen Logik und Grundlagen der Mathematik, Volume 6, Pages 66-92, 1960.

[43] A. Boudet, H. Comon. *Diophantine equations, Presburger arithmetic and finite automata*, In Proceedings of International Colloquium on Trees in Algebra and Programming (CAAP), Pages 30-43, 1996.

[44] P. Wolper, B. Boigelot. *On the construction of automata from linear arithmetic constraints*, In Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Pages 1-19, 2000.

[45] M. Garey, D. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*, W. H. Freeman and Company, 1979.

[46] B. Dutertre, L. de Moura. *Integrating simplex with DPLL(T)*, Technical Report, Computer Science Laboratory (CSI), Stanford Research Institute (SRI), 2006.

[47] C. Scholl, S. Disch, F. Pigorsch, S. Kupferschmid. *Computing optimized representations of non-convex polyhedra by detection and removal of redundant linear constraints*, In Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Pages 383-397, 2009.

[48] A. Mishchenko, S. Chatterjee, R. Jiang, R. Brayton. *FRAIGs: A unifying representation for logic synthesis and verification*, Technical Report, EECS Department, UC Berkeley, 2005.

[49] N. Bjørner. *Linear quantifier elimination as an abstract decision procedure*, In Proceedings of International Joint Conference on Automated Reasoning (IJCAR), Pages 316-330, 2010.

[50] T. Nipkow. *Linear quantifier elimination*, In Proceedings of International Joint Conference on Automated Reasoning (IJCAR), Pages 18-33, 2008.

[51] R.E. Bryant. *Graph-based algorithms for boolean function manipulation*, In IEEE Transactions on Computers, Volume 35, Issue 8, Pages 677-691, 1986.

[52] D. Monniaux. *Quantifier elimination by lazy model enumeration*, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 585-599, 2010.

[53] A. Phan, N. Bjorner, D. Monniaux. *Anatomy of alternating quantifier satisfiability (work in progress)*, In Proceedings of SMT Workshop at International Joint Conference on Automated Reasoning (SMT@IJCAR), Pages 120-130, 2012.

[54] S. Lahiri, R. Nieuwenhuis, A. Oliveras. *SMT techniques for fast predicate abstraction*, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 424-437, 2006.

[55] L. de Moura, N. Bjorner. *Relevancy propagation*, Technical Report TR-2007-140, Microsoft Research, 2007.

[56] D. Déharbe, P. Fontaine, D. Le Berre, B. Mazure. *Computing prime implicants*, In Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), Pages 46-52, 2013.

[57] A. Niemetz, M. Preiner, A. Biere. *Turbo-charging lemmas on demand with don't care reasoning*, In Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), Pages 179-186, 2014.

[58] A. Komuravelli, A. Gurfinkel, S. Chaki. *SMT-based model checking for recursive programs*, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 17-34, 2014.

[59] N. Bjorner, M. Janota. *Playing with quantified satisfaction*, In Proceedings of International Conferences on Logic for Programming, Artificial Intelligence and Reasoning (LPAR) - Short Presentations, Pages 15-27, 2015.

[60] S. Owre, J. Rushby, N. Shankar. *PVS: A prototype verification system*, In Proceedings of International Conference on Automated Deduction (CADE), Pages 748-752, 1992.

[61] J. Ax, S. Kochen. *Diophantine problems over local fields II. A complete set of axioms for p-adic number theory*, In American Journal of Mathematics, Volume 87, Issue 3, Pages 631-648, 1965.

[62] P. Cohen. *Decision procedures for real and p-adic fields*, In Communications in Pure and Applied Logic, Volume 25, Pages 213-231, 1969.

[63] D. Babic, M. Musuvathi. *Modular arithmetic decision procedure*, Technical Report TR-2005-114, Microsoft Research, 2005.

[64] N. Tew, P. Kalla, N. Shekhar, S. Gopalakrishnan. *Verification of arithmetic datapaths using polynomial function models and congruence solving*, In Proceedings of International Conference on Computer-Aided Design (ICCAD), Pages 122-128, 2008.

[65] D. Cyrluk, M. Möller, H. Rueß. *An efficient decision procedure for the theory of fixed-sized bit-vectors*, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 60-71, 1997.

[66] N. Bjørner, M. Pichora. *Deciding fixed and non-fixed size bit-vectors*, In Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Pages 376-392, 1998.

[67] A. Griggio. *Effective word-level interpolation for software verification*, In Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), Pages 28-36, 2011.

[68] W. Pugh. *The Omega Project: Frameworks and algorithms for the analysis and transformation of scientific programs*, `www.cs.umd.edu/projects/omega`.

[69] S. Davidson. *Characteristics of the ITC'99 benchmark circuits*, In Proceedings of IEEE International Test Synthesis Workshop (ITSW), 1999, `cerc.utexas.edu/itc99-benchmarks/bench.html`.

[70] N. Eén, A. Biere. *Effective preprocessing in SAT through variable and clause elimination*, In Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT), Pages 61-75, 2005.

[71] M. Järvisalo, A. Biere, M. Heule. *Blocked clause elimination*, In Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Pages 129-144, 2010.

[72] M. Heule, M. Järvisalo, F. Lonsing, M. Seidl, A. Biere. *Clause elimination for SAT and QSAT*, In Journal of Artificial Intelligence Research, Volume 53, Issue 1, Pages 127-168, 2015.

[73] B. Kiesl, M. Seidl, H. Tompits, A. Biere. *Super-blocked clauses*, In Proceedings of International Joint Conference on Automated Reasoning (IJCAR), Pages 45-61, 2016.

[74] C. Barrett, D. Dill, J. Levitt. *A decision procedure for bit-vector arithmetic*, In Proceedings of ACM/IEEE Design Automation Conference (DAC), Pages 522-527, 1998.

[75] N.S. Szabo, R.I. Tanaka. *Residue arithmetic and its applications to computer technology*, McGraw-Hill, 1967.

[76] J. A. Howell, R. T. Gregory. *An algorithm for solving linear algebraic equations using residue arithmetic I*, In BIT Numerical Mathematics, Volume 9, Issue 3, Pages 200-224, 1969.

[77] M. Muller-Olm, H. Seidl. *Analysis of modular arithmetic*, In ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 29, Issue 5, 2007.

[78] C.Y. Huang, K.T. Cheng. *Assertion checking by combined word-level ATPG and modular arithmetic constraint-solving techniques*, In Proceedings of ACM/IEEE Design Automation Conference (DAC), Pages 118-123, 2000.

[79] V. Ganesh, D. Dill. *A decision procedure for bit-vectors and arrays*, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 519-531, 2007.

[80] H. Jain, E. M. Clarke, O. Grumberg. *Efficient Craig interpolation for linear diophantine (dis)equations and linear modular equations*, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 254-267, 2008.

[81] G. Gange, H. Sondergaard, P. Stuckey, P. Schachte. *Solving difference constraints over modular arithmetic*, In Proceedings of International Conference on Automated Deduction (CADE), Pages 215-230, 2013.

[82] R. Brummayer, A. Biere. *Boolector: An efficient SMT solver for bit-vectors and arrays*, In Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Pages 174-177, 2009.

[83] R. Brinkmann, R. Drechsler. *RTL-datapath verification using integer linear programming*, In Proceedings of IEEE VLSI Design Conference, Pages 741-746, 2002.

[84] R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, Z. Hanna, A. Nadel, A. Palti, R. Sebastiani. *A lazy and layered SMT(BV) solver for hard industrial*

verification problems, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 547-560, 2007.

[85] L. Hadarean, K. Bansal, D. Jovanovic, C. Barret, C. Tinelli. *A tale of two solvers: Eager and lazy approaches to bit-vectors*, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 680-695, 2014.

[86] A. Gotlieb, M. Leconte, B. Marre. *Constraint solving on modular integers*, In Proceedings of Ninth International Workshop on Constraint Modelling and Reformulation (ModRef) co-located with International Conference on Principles and Practice of Constraint Programming (CP), 2010.

[87] M. Veanes, N. Bjørner, L. Nachmanson, S. Bereg. *Monadic decomposition*, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 628-645, 2014.

[88] B. Selman, H. Kautz. *Planning as satisfiability*, In Proceedings of European Conference on Artificial Intelligence (ECAI), Pages 359-363, 1992.

[89] D. Brand. *Verification of large synthesized designs*, In Proceedings of International Conference on Computer-Aided Design (ICCAD), Pages 534-537, 1993.

[90] I. Lynce, J. Marques-Silva. *Efficient haplotype inference with boolean satisfiability*, In Proceedings of National Conference on Artificial Inference (AAAI), Pages 104-109, 2006.

[91] M. Davis, G. Longemann, D. Loveland. *A machine program for theorem proving*, In Communications of the ACM, Volume 5, Issue 7, Pages 394-397, 1962.

[92] F. Pigorsch, C. Scholl, S. Disch. *Advanced unbounded CTL model checking using AIGs, BDD sweeping, and quantifier scheduling*, In Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), Pages 89-96, 2006.

[93] M. Benedetti. *Skizzo: A suite to evaluate and certify QBFs*, In Proceedings of International Conference on Automated Deduction (CADE), Pages 369-376, 2005.

[94] T. Jussila, A. Biere, C. Sinz, D. Kroening, C. Wintersteiger. *A first step towards a unified proof checker for QBF*, In Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT), Pages 201-214, 2007.

[95] M. Narizzano, C. Peschiera, L. Pulina, A. Tacchella. *Evaluating and certifying QBFs: A comparison of state-of-the-art tools*, In AI Communications, Volume 22, Issue 4, Pages 191-210, 2009.

[96] Y. Yu, S. Malik. *Validating the result of a quantified boolean formula (QBF) solver: Theory and practice*, In Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC), Pages 1047-1051, 2005.

[97] M. J. H. Heule, M. Seidl, A. Biere. *A unified proof system for QBF preprocessing*, In Proceedings of International Joint Conference on Automated Reasoning (IJCAR), Pages 91-106, 2014.

[98] C. Wintersteiger, Y. Hamadi, L. de Moura. *Efficiently solving quantified bit-vector formulas*, In Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), Pages 239-246, 2010.

[99] Berkeley Logic Synthesis and Verification Group. *ABC: A system for sequential synthesis and verification*, http://www.eecs.berkeley.edu/ alanmi/abc/

[100] G. Tseitin. *On the complexity of derivations in the propositional calculus*, In Studies in Mathematics and Mathematical Logic, Part II, Pages 115-125, 1968.

[101] SMTLib website, http://goedel.cs.uiowa.edu/smtlib/.

[102] STP website, http://sites.google.com/site/stpfastprover/.

[103] B. Jobstmann, A. Griesmayer, R. Bloem. *Program repair as a game*, In Proceedings of International Conference on Computer Aided Verification (CAV), Pages 226-238, 2005.

[104] P. J. Ramadge, W. M. Wonham. *Supervisory control of a class of discrete event processes*, In Society for Industrial and Applied Mathematics (SIAM) Journal on Control and Optimization, Volume 25, Issue 1, Pages 206-230, 1987.

[105] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. *Counterexample-guided abstraction refinement for symbolic model checking*, In Journal of the ACM (JACM), Volume 50, Issue 5, Pages 752-794, 2003.

[106] DepQBF website, http://lonsing.github.io/depqbf/.

[107] HWMCC10 website, `fmv.jku.at/hwmcc10/benchmarks.html`.

[108] Bloqqer website, `fmv.jku.at/bloqqer`.

[109] QBFcert website, `fmv.jku.at/qbfcert`.

[110] R. Bruttomesso, N. Sharygina. *A scalable decision procedure for fixed-width bit-vectors*, In Proceedings of International Conference on Computer-Aided Design (ICCAD), Pages 13-20, 2009.

[111] G. Nelson, D. C. Oppen. *Simplification by cooperating decision procedures*, In ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 1, Issue 2, Pages 245-257, 1979.