Satisfiability Modulo Theories for finite domains

Auteur : Dasnois, Louis
Promoteur(s) : Fontaine, Pascal
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil en informatique, à finalité spécialisée en "computer systems security"
Année académique : 2022-2023
URI/URL : http://hdl.handle.net/2268.2/18345

University of Liège
School of Engineering and Computer Science

Satisfiability Modulo Theories for Finite Domains

Master's thesis completed in order to obtain the degree of
Master of Science in Computer Science and Engineering
by Dasnois Louis

Supervisor:
Fontaine Pascal

Academic Year 2022-2023

Abstract

This thesis investigates strategies to improve and extend modern SMT solving procedures to effectively handle problems involving finite domains, whether they are explicitly defined or concealed within the problem's encoding. The research comprises both theoretical analyses and empirical evaluations. The central contributions are twofold:

First, a quantifier elimination strategy is developed. The research reveals that quantifiers significantly hinder solver efficiency when applied to finite domains, using Sudoku as an example. An algorithm is introduced to automatically detect "effective finite domains" in quantified Skolem formulas over integers, and eliminate quantifiers through exhaustive instantiation. A prototype implementation is made, showing this procedure improves solver performance and can be used even with solvers lacking quantifier support.

Second, the thesis explores the theory of uninterpreted functions with domain cardinality constraints. It establishes the NP-completeness of the satisfiability problem for a set of literals in this theory. SAT-based algorithms extending classical congruence closure are proposed, ensuring the efficiency of congruence closure for large or infinite domains while transitioning to SAT solvers for smaller domains. The proposed algorithms await implementation and validation within an SMT framework. Notably, addressing the loss of conflict set generation and equality deduction when SAT is employed remains an open challenge for future research.

Acknowledgements

I would like to express my heartfelt gratitude to Pascal Fontaine, my advisor, for his invaluable guidance and support throughout this research. He introduced me to the beautiful fields of logic and automated reasoning, answered each of my questions, helped me out with hardware issues, and kept me on track for completing this work.

I extend my appreciation to Tanja Schindler for her assistance, paper recommendations, and meticulous proofreading of some sections. I am also grateful to her and Bernard Boigelot for accepting to evaluate this work.

I want to acknowledge the dedicated professors and assistants who have enriched my knowledge over the past five years.

Lastly, my deepest thanks go to my family and friends for their continued support during my academic journey.

Contents

1 Introduction **5**

2 Background **8**
2.1 Propositional logic 8
2.2 SAT solving 11
2.3 First-order logic 12
2.4 Herbrand theory 15
2.5 Satisfiability modulo theories 17
2.5.1 Theory solver 17
2.5.2 Ground SMT solving 18
2.5.3 Theory combination 18
2.5.4 Quantifiers 20

3 Sudoku in SMT **21**
3.1 Encoding 21
3.1.1 Quantifier-free encoding 22
3.1.2 Propositional encoding 22
3.1.3 Encoding using quantifiers 23

4 Performance of existing solvers **24**
4.1 Satisfiable case 24
4.2 Unsatisfiable case 26
4.3 Larger Sudokus 27

5 A first approach: dealing with quantifiers **29**
5.1 Detecting finite domains over integers 29
5.2 Removing quantifiers with exhaustive instantiation 33
5.3 Extension to multiple formulas quantified over multiple variables 33
5.3.1 Multiple formulas 33
5.3.2 Formulas quantified over multiple variables 34
5.4 Implementation and results 35

6 Theory of uninterpreted functions over finite domains 38
6.1 Classical congruence closure 38
6.2 Finite domain implications 42
6.3 Application to Sudoku 43
6.4 NP-completeness 44
6.5 Approaching the problem using SAT 45
6.6 Algorithm for a single sort 46
6.6.1 SAT encoding 48
6.7 Extensions for multiple sorts 50
6.7.1 Isolating the domains 50
6.7.2 Lazy evaluation of inter-domain constraints 51
6.7.3 Problems with isolating the domains 54
6.7.4 Full SAT encoding of the multi-sorted problem 54
6.7.5 Features of the theory solver 56

7 Conclusions 58

A Source code 60

Chapter 1

Introduction

In the realm of computational logic and formal verification, the task of determining the satisfiability of logical formulas plays a key role in various applications, ranging from software and hardware verification to automated reasoning and constraint solving. SAT solving, a technique rooted in propositional logic, is a fundamental approach employed to tackle such satisfiability problems by transforming them into propositional encodings. The expressivity of propositional logic is however limited, and SAT may not be the best approach for problems for which more specialized, efficient algorithms exist, such as linear arithmetic . The Satisfiability Modulo Theories (SMT) framework has emerged as a powerful approach to address this problem by combining first-order logic with specialized theories tailored to specific domains.

SMT is a rapidly growing field, opening a wide area of research. Each year, an international competition, SMT-COMP [1], is organized. Standardized benchmarks [2] are used to evaluate and compare the performance of different SMT solvers supporting the SMT-LIB input format [3].

SMT has a wide range of applications, including hardware and software verification [4][5], automated theorem proving [6], model checking [7], constraint satisfaction problems [8], and even applications in security [9]: SMT can be used for symbolic execution [10], allowing the detection of bugs and potential vulnerabilities, or helping in the reverse engineering and analysis of malware. It also has uses in cryptography, through the verification of cryptographic protocols [11], or using the constraint solving capabilities for cryptanalysis [12].

The research in thesis falls in the scope of the ANR (Agence Nationale de la Recherche, France) project BLaSST [1] (Enhancing B Language Reason-

[1] `https://merz.gitlabpages.inria.fr/blasst/`

ers Using SAT and SMT Techniques), in collaboration with ClearSy, CRIL (Lens), and Inria Nancy.

It was found that, during verification efforts, the proof obligations arising from industrial models often contain problems which are combinatorial in nature. Usually, this is the result of the problem being defined on some underlying finite domain. For such problems, SAT-based techniques, reducing the problem to a propositional encoding, are usually very effective and efficient. However, propositional encodings have limited expressivity, and can be quite tricky or unintuitive to work with. Allowing those finite domain problems to be encoded in the language of first-order logic makes them easier to express, and also allows for the encoding of problems that may be impossible to express in purely propositional logic, for example because some component of the problem also relies on an infinite domain, with an infinite amount of constraints to encode. It can also be the case that the problem contains a mix of combinatorial, finite domain constraints, and constraints that can be more efficiently reasoned with using a specialized solver. For example, linear arithmetic problems are often best solved with methods based on the simplex algorithm. For such problems, the exclusive use of SAT may not be the most efficient approach: combining different specialized techniques, including SAT, such that each part of the problem can be reasoned about efficiently can lead to better results. This is exactly the goal of Satisfiability Modulo Theories. Unfortunately, SMT solvers can often struggle working with finite domains, and in some cases lead to poor performances compared to the equivalent SAT approach.

This lack in competitiveness can be due to multiple reasons. One of them can be the inefficiencies between the different theory solvers and the underlying SAT solver. If a theory solver fails to give concise explanations of why an assignment fails, or if conflicts involve many literals, the SAT solver does not learn much from each call to the theory solver, and is left to check every possible assignment exhaustively. There can also be inefficiencies in the exchange of information between the different theory solvers, when multiple theories are involved. In those cases, it may be beneficial to include some (for example SAT-based) combinatorial reasoning at the level of the theory solver itself, in order to come up with a satisfying assignment directly.

Another possibility for SMT struggling with finite domains is that the finiteness of the domain is hidden within the specifics of the encoding. Typically, finite domains only have a finite number of constraints applied to them, and those constraints can thus be expressed exhaustively. Still, those constraints often being repetitive, it may be more natural to make use of universally quantified formulas to encode them. Doing so usually hides the fact that the domain is finite from the solver, as the constraints now apply

to an infinite number of elements. In most cases, the domain can still be considered finite, because the constraints are only really restrictive for some finite amount of elements. This is however not always obvious to the solver, and it can fail to apply appropriate techniques to solve the problem because it is unable to recognize that the domain is inherently finite. In such cases, it would be helpful to have way to detect the presence of a finite domain, so that appropriate techniques can be applied to solve the problem efficiently.

The goal of this work is to come up with ways to extend and improve modern SMT-solving procedures in order to better handle problems that range over some finite domain, be it explicitly defined, or hidden within the encoding of the problem. To do that, different approaches are studied and compared theoretically. Their correctness is established, as well as their completeness and complexity whenever possible. Some approaches are also tested empirically on the well-known Sudoku problem, a good representative of combinatorial, finite domain satisfiability problems.

The next chapters are organized as follows: Chapter 2 introduces the notation, gives some theoretical background on logic and satisfiability, and goes over the basics of how modern SMT solvers work, as well as some of the more advanced notions which are relevant to this work. Chapter 3 introduces the Sudoku problem, how it is relevant to satisfiability problems over finite domains, and the multiple ways it can be encoded as a satisfiability problem. The Sudoku example is used throughout this work as a motivating example, as well as to show how the different approaches work in practice. Chapter 4 contains a quick analysis of how current solvers handle Sudokus in their different encodings, illustrating that there is room for improvement especially for quantified encodings. Chapter 5 explores ways to automatically detect instances of finite domain problems in quantified encodings on integers, and how to automatically convert them to more efficient quantifier-free encodings, capturing the finite nature of the problem. Finally, Chapter 6 attempts to extend the congruence closure algorithm, commonly used for reasoning on equality and uninterpreted functions, to domains with a finite cardinality constraint. This way, equalities imposed by the finite size of the domain can be directly deduced at the level of the theory solver for uninterpreted functions. After showing that the problem is NP-complete, a way of combining a SAT solver with the classical congruence closure algorithm is proposed, leading to an efficient procedure for solving Sudokus (and other related problems) encoded in SMT.

Chapter 2

Background

This chapter introduces the notations, and gives a summary of the important notions from propositional and first-order logic which are used throughout this work. It also briefly explains how SAT solving works, and gives an overview of the inner workings of SMT solvers. Definitions, theorems, and explanations are adapted from [13][14][15][16]. Proofs (or links to proofs) can also be found there, but are not copied here as they would take up space and are not particularly relevant for this work.

2.1 Propositional logic

Propositional logic deals with formulas composed of boolean propositions and connectives. It is one of the simplest forms of logic.

Definition 1 (Propositional formula)**.** A *propositional formula* is a string of symbols combining boolean *propositions* and *connectives*, in the following way:

- a proposition, usually denoted with a (possibly indexed) lowercase letter (e.g. p, q) is a formula;
- $\bot$, $\top$ are formulas;
- if φ is a formula, $\neg\varphi$ is a formula;
- if φ and ψ are formulas, $(\varphi \circ \psi)$, where $\circ$ can be $\vee$, $\wedge$, $\Rightarrow$ or $\Leftrightarrow$, is a formula.

$\bot$ and $\top$ represent formulas that are respectively always false and always true. $\neg$ represents the negation of a formula. $\vee$ represents the disjunction

(inclusive or) of two formulas, $\wedge$ represents their conjunction (and), $\Rightarrow$ represents implication, and $\Leftrightarrow$ represents equivalence. Parentheses can be omitted when there is no ambiguity. In particular, operators $\wedge$ and $\vee$ are associative: $p \wedge q \wedge r$ is the same as either $p \wedge (q \wedge r)$ or $(p \wedge q) \wedge r$ (and similarly for $\vee$). When parentheses are missing, $\neg$ takes precedence over other operators. Parentheses can also be added to improve readability.

Definition 2 (Interpretation of a (set of) propositional formula(s))**.** An interpretation of a (set of) formula(s) containing propositions $p_1, \ldots, p_n$ is a function $\mathcal{I} : \{p_1, \ldots, p_n\} \longrightarrow \{T, F\}$ assigning a truth value to each proposition in the (set of) formula(s). A truth value is assigned to a formula φ according to the following rules:

- $\mathcal{I}[\bot] = F$, $\mathcal{I}[\top] = T$;
- $\mathcal{I}[\neg\varphi] = F$ if $\mathcal{I}[\varphi] = T$, T otherwise;
- $\mathcal{I}[\varphi_1 \vee \varphi_2] = F$ if $\mathcal{I}[\varphi_1] = F$ and $\mathcal{I}[\varphi_2] = F$, T otherwise;
- $\mathcal{I}[\varphi_1 \wedge \varphi_2] = T$ if $\mathcal{I}[\varphi_1] = T$ and $\mathcal{I}[\varphi_2] = T$, F otherwise;
- $\mathcal{I}[\varphi_1 \Rightarrow \varphi_2] = F$ if $\mathcal{I}[\varphi_1] = T$ and $\mathcal{I}[\varphi_2] = F$, T otherwise;
- $\mathcal{I}[\varphi_1 \Leftrightarrow \varphi_2] = T$ if $\mathcal{I}[\varphi_1] = \mathcal{I}[\varphi_2]$, F otherwise;

An interpretation makes a set of formulas true if and only if it makes every formula in the set true.

Definition 3 (Model)**.** An interpretation $\mathcal{I}$ of a formula φ is a *model* of φ (denoted $\mathcal{I} \models \varphi$) if and only if it makes the formula true: $\mathcal{I}[\varphi] = T$. An interpretation of a set of formulas S is a model of S (denoted $\mathcal{I} \models S$) if and only if $\mathcal{I}[\varphi] = T$ for all $\varphi \in S$, i.e., if it is a model of every formula in the set.

Definition 4 (Satisfiability)**.** A (set of) formula(s) is said to be *satisfiable* if and only if it has at least one model. Otherwise, it is *unsatisfiable*.

Definition 5 (Validity)**.** A formula φ is said to be *valid* if and only if every interpretation is a model of φ. This is denoted $\models \varphi$.

Theorem 1. *A formula is valid if and only if its negation is unsatisfiable.*

Definition 6 (Logical consequence)**.** A formula φ is a *logical consequence* of a set of formulas S (denoted $S \models \varphi$) if and only if every model of S is a model of φ.

Theorem 2 (Deduction theorem)**.** *Given a formula φ and a finite set of formulas $S = \{\psi_1, \ldots, \psi_n\}$, the following statements are equivalent:*

- *$S \models \varphi$, i.e. φ is a logical consequence of S;*
- *$S \cup \{\neg\varphi\} \models \bot$, i.e. $S \cup \{\neg\varphi\}$ is unsatisfiable;*
- *$\models (\psi_1 \wedge \ldots \wedge \psi_n) \Rightarrow \varphi$, i.e. $(\psi_1 \wedge \ldots \wedge \psi_n) \Rightarrow \varphi$ is valid.*

If S is infinite, only the first two statements are equivalent.

Theorem 3. *Given a set of formulas S and a logical consequence φ of S ($S \models \varphi$), S is satisfiable if and only if $S \cup \{\varphi\}$ is satisfiable.*

Definition 7 (Logical equivalence)**.** Two formulas φ and ψ are *logically equivalent* (denoted $\varphi \longleftrightarrow \psi$) if they have the same models, i.e., if $\mathcal{I}[\varphi] = \mathcal{I}[\psi]$ for every interpretation $\mathcal{I}$.

Theorem 4. *Two formulas φ and ψ are logically equivalent ($\varphi \longleftrightarrow \psi$) if and only if $\models \varphi \Leftrightarrow \psi$.*

Theorem 5. *Every n-ary connective ($n > 2$) can be simulated with binary connectives and negations.*

Definition 8 (Literal, clause, cube)**.** A *literal* is either a proposition, or the negation of a proposition (e.g. p, q, $\neg r$). A *clause* is a disjunction ($\vee$) of literals (e.g. $q \vee \neg p \vee r$). A *cube* is a conjunction ($\wedge$) of literals (e.g. $\neg p \wedge \neg q$).

Definition 9 (Conjunctive normal form)**.** A formula is in *conjunctive normal form* (CNF) if it is a conjunction of one or more clauses.

Definition 10 (Disjunctive normal form)**.** A formula is in *disjunctive normal form* (DNF) if it is a disjunction of one or more cubes.

Theorem 6. *Every formula is logically equivalent to a CNF and to a DNF.*

For example, $\neg(p \Rightarrow q) \vee r$ is equivalent to $(p \wedge \neg q) \vee r$ (DNF), and $(p \vee r) \wedge (\neg q \vee r)$ (CNF).

Theorem 7 (Tseitin transformation [17])**.** *Every formula φ can be transformed in linear time into an equisatisfiable CNF, i.e., a CNF which is satisfiable if and only if φ is satisfiable.*

Theorem 8 (Compactness theorem)**.** *A (possibly infinite) set of formulas is satisfiable if and only if all its finite subsets are satisfiable.*

In particular, this last theorem guarantees that any unsatisfiable set of formulas, even infinite, has a finite unsatisfiable subset.

2.2 SAT solving

Definition 11 (Satisfiability problem)**.** The *satisfiability problem* in propositional logic (SAT) is the problem of determining whether a (finite set of) propositional formula(s) is satisfiable.

The SAT problem is a well-known NP-complete problem [18]. A decision procedure for the propositional satisfiability problem is commonly called a SAT solver. Despite the difficulty of NP-complete problems, modern SAT solvers are able to efficiently solve many common instances of the problem, up to sizes with millions of propositions. This makes SAT solvers very versatile tools that are used for many applications in fields such as hardware and software verification, automated reasoning, and artificial intelligence among other things. The practice of encoding an NP problem in SAT before giving it to a SAT solver is a generic and common approach, which often results in an efficient procedure for solving said problem.

Modern, state-of-the-art SAT solvers use an algorithm called conflict-driven clause learning (CDCL) [19]. The details of the algorithm will not be discussed here, as they are not particularly relevant for this work, but here is a brief overview. The algorithm expects an input formula in CNF, generally given to the solver in the DIMACS format [20]. An equisatisfiable CNF can efficiently be derived from any arbitrary formula using the Tseitin transformation [17]. The CDCL algorithm keeps track of a partial assignment of truth values to the propositions using a stack. It does a few operations:

- Propagation: when every literal in a clause is assigned a negative value except one, this last literal must be assigned true. If all the literals in a clause are false, a conflict is reached.
- Decide: when there is nothing to propagate, no conflict, and only a partial assignment, a value must be decided arbitrarily for one of the remaining propositions.
- Conflict analysis: when a conflict is reached, a new clause can be learned, which would have avoided taking the decision that lead to this conflict.
- Backtracking: after a conflict is reached and analysed, the last steps are undone (literals are popped from the stack) until the conflict clause learned in analysis contains one unassigned literal. Then, propagation can resume.

The details of conflict analysis are not given here, but it is a key part of the algorithm. Generally, the smaller a conflict clause is, the more useful it is,

as it reduces the number of possibilities to consider. Many additional techniques are used to make SAT solvers more efficient. They include heuristics for taking decisions, clause removal, conflict clause minimization, and many other things.

2.3 First-order logic

Applications in many fields, notably software verification or automated theorem proving, are interested in reasoning beyond propositional logic. Indeed, the expressivity of propositional logic is quite limited, and determining the satisfiability of formulas in more expressive logics proves very useful. First-order logic (FOL) expands on propositional logic and allows for much greater expressivity. Propositions become predicates, meaning their truth values can depend on a certain number of terms. Those terms can also depend on other terms (i.e., there are functions). Last but not least, independent variables can also be quantified either existentially or universally. The combination of functions and quantifiers makes the satisfiability problem in first-order logic is only semi-decidable: unsatisfiability can be deduced in finite time by an algorithm, but if the formula is satisfiable, the algorithm may never terminate. In SMT, we consider *sorted* first-order logic, where terms can be of different sorts, or types, corresponding to different domains.

Definition 12 (Many-sorted first-order language)**.** A *many-sorted first-order language* is a tuple $\mathcal{L} = (\mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, r, d)$, comprising:

- a countable non-empty set of *sorts* $\mathcal{S}$;
- a countable set of *variables* $\mathcal{V} = \bigcup_{\tau \in \mathcal{S}} \mathcal{V}_\tau$, where $\mathcal{V}_\tau$ are the disjoint countable sets of variables of sort τ;
- a countably infinite set of *function symbols* $\mathcal{F}$;
- a countably infinite set of *predicate symbols* $\mathcal{P}$;
- a function $r : \mathcal{F} \cup \mathcal{P} \longrightarrow \mathbb{N}$, assigning an *arity* to each function and predicate symbol;
- a function $d : \mathcal{F} \cup \mathcal{P} \longrightarrow \mathcal{S}^*$, assigning a sort in $S^{r(f)+1}$ to each function symbol $f \in \mathcal{F}$, and sort in $S^{r(p)}$ to each predicate symbol $p \in \mathcal{P}$.

Nullary predicates are *propositions* and nullary functions are *constants*. The *signature* of $\mathcal{L}$ is the tuple $(\mathcal{S}, \mathcal{F}, \mathcal{P}, r, d)$.

Definition 13 (First-order term)**.** Given a sort τ and a first-order language $\mathcal{L} = (\mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, r, d)$, the set of *$\tau$-terms* is the smallest set such that:

- each $x \in \mathcal{V}_\tau$ is a τ-term;
- for each function symbol $f \in \mathcal{F}$ of sort $(\tau_1, \ldots, \tau_n, \tau)$, $f(t_1, \ldots, t_n)$ is a τ-term if $t_1, \ldots, t_n$ are $\tau_1, \ldots, \tau_n$-terms respectively.

$Sort(t) = \tau$ if t is a τ-term. A *ground term* is a term which does not depend on any variables.

Definition 14 (Atomic formula)**.** Given a first-order language $\mathcal{L} = (\mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, r, d)$, the set of *atomic formulas*, or *atoms*, is the smallest set such that:

- for each predicate symbol $p \in \mathcal{P}$ of sort $(\tau_1, \ldots, \tau_n)$, $p(t_1, \ldots, t_n)$ is an atomic formula if $t_1, \ldots, t_n$ are $\tau_1, \ldots, \tau_n$-terms respectively;
- for each pair t, t' of τ-terms, $t = t'$ is an atomic formula.

A *literal* is either an atomic formula, or the negation of an atomic formula. $\top$ and $\bot$ correspond to universally true and false atoms, respectively.

Definition 15 (First-order formula)**.** Given a first-order language $\mathcal{L} = (\mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, r, d)$, the set of *formulas*, is the smallest set such that:

- atomic formulas are formulas;
- if φ is a formula, $\neg\varphi$ is a formula;
- if φ and ψ are formulas, $(\varphi \circ \psi)$, where $\circ$ can be $\vee$, $\wedge$, $\Rightarrow$ or $\Leftrightarrow$, is a formula;
- if φ is a formula and $x \in \mathcal{V}$ is a variable, $\forall x\ \varphi$ and $\exists x\ \varphi$ are formulas.

Definition 16 (Scope, free and bound variables)**.** The *scope* of a quantifier (or its variable) is the formula under the quantifier. In $\forall x\ \varphi$ or $\exists x\ \varphi$, the scope of x is φ, x is *quantified* in $\forall x$ or $\exists x$, every occurrence of x in φ is *bound*. A variable occurence is *free* when it is neither bound nor quantified. A formula is *closed* if it does not contain any free variables.

Similarly to propositional logic, parentheses can be added or removed to improve readability. A dot after a quantified variable stands for an opening parenthesis closing at the end of the formula. $\forall x, y\ \varphi$ stands for $\forall x \forall y\ \varphi$. For example, $\forall x, y.\, x = f(y) \ \vee\ (p(x) \Rightarrow p(y))$ stands for $\forall x \forall y\ (x = f(y) \ \vee\ (p(x) \Rightarrow p(y)))$.

Definition 17 (First-order interpretation)**.** An *interpretation* of a formula in a many-sorted first-order language $\mathcal{L} = (\mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, r, d)$, is a tuple $\mathcal{I} = (D, I_{\mathcal{V}}, I_{\mathcal{F}}, I_{\mathcal{P}})$, where:

- D assigns a non-empty domain $\mathcal{D}_\tau$ (set) to each sort $\tau \in \mathcal{S}$. Those domains are not necessarily distinct;
- $I_{\mathcal{V}}$ assigns an element in $\mathcal{D}_\tau$ to each variable $x \in \mathcal{V}_\tau$;
- $I_{\mathcal{F}}$ assigns a function $\mathcal{D}_{\tau_1} \times \ldots \times \mathcal{D}_{\tau_n} \longrightarrow \mathcal{D}_\tau$ to each function symbol $f \in \mathcal{F}$ of sort $(\tau_1, \ldots, \tau_n, \tau)$;
- $I_{\mathcal{P}}$ assigns a function $\mathcal{D}_{\tau_1} \times \ldots \times \mathcal{D}_{\tau_n} \longrightarrow \{T, F\}$ to each predicate symbol $p \in \mathcal{P}$ of sort $(\tau_1, \ldots, \tau_n)$;

Given an interpretation $\mathcal{I} = (D, I_{\mathcal{V}}, I_{\mathcal{F}}, I_{\mathcal{P}})$, $\mathcal{I}_{x/d}$ represents the interpretation $(D, I'_{\mathcal{V}}, I_{\mathcal{F}}, I_{\mathcal{P}})$ where $I'_{\mathcal{V}}[x] = d$ and $I'_{\mathcal{V}}[y] = I_{\mathcal{V}}[y]$ for each variable y other than x. An interpretation assigns a value $\mathcal{I}[t] \in \mathcal{D}_\tau$ to each τ-term t:

- if t is a free variable, $\mathcal{I}[t] = I_{\mathcal{V}}[t]$;
- if $t = f(t_1, \ldots, t_n)$, $\mathcal{I}[t] = I_{\mathcal{F}}[f](\mathcal{I}[t_n], \ldots, \mathcal{I}[t_n])$.

. Similarly, it assigns a truth value to each formula:

- for predicate atoms, $\mathcal{I}[p(t_1, \ldots, t_n)] = I_{\mathcal{P}}[p](\mathcal{I}[t_n], \ldots, \mathcal{I}[t_n])$;
- for equality atoms, $\mathcal{I}[t = t'] = T$ if $\mathcal{I}[t] = \mathcal{I}[t']$, F otherwise;
- $\mathcal{I}[\forall x\ \varphi] = T$ if $\mathcal{I}_{x/d}[\varphi] = T$ for each $d \in \mathcal{D}_{Sort(x)}$, F otherwise;
- $\mathcal{I}[\exists x\ \varphi] = T$ if $\mathcal{I}_{x/d}[\varphi] = T$ for at least one $d \in \mathcal{D}_{Sort(x)}$, F otherwise;
- the rest is interpreted as for propositional logic.

Definitions 3-7 for models, satisfiability, validity, logical consequence, and logical equivalence in propositional logic hold for first-order logic. Theorems 1-4 (including notably the deduction theorem) hold as well.

Theorem 9 (Undecidability of first-order logic)**.** *The problem of checking whether an arbitrary first-order logic formula is satisfiable or not is undecidable.*

Theorem 10 (Semi-decidability of first-order logic)**.** *There exists a sound procedure to check the satisfiability of (finite or denumerable sets of) first-order formulas, that always terminates for unsatisfiable (set of) formulas.*

2.4 Herbrand theory

Theorem 10 is a result stemming from Herbrand theory. Herbrand theory gives a multitude of results which are fundamental for satisfiability solving in first-order formulas. The most important ones are given below.

Definition 18 (Prenex form)**.** A formula is in *prenex form* if it is written as

$$Q_1 x_1 \dots Q_n x_n(\varphi)$$

where $Q_1, \dots, Q_n \in \{\exists, \forall\}$, $x_1, \dots, x_n \in \mathcal{V}$, and φ is quantifier-free. The formula φ is the *matrix* of the prenex form.

Theorem 11. *Any first-order formula can be converted to a logically equivalent prenex form.*

This is done through a series of equivalence-preserving steps which bring the quantifiers to the top level.

Definition 19 (Skolem form)**.** A formula is in *Skolem form* if it is in prenex form without existential quantifiers.

Theorem 12. *Any first-order formula can be converted to an equisatisfiable Skolem form.*

The process of converting a formula to an equisatisfiable Skolem form is known as *Skolemization*. Because of the nice properties of formulas in Skolem form (notably in Theorems 13 and 14), it is often used as a first step in solvers. Skolemization works by introducing *Skolem functions* to replace existentially quantified variables. For example, formula $\forall x, y \exists z \varphi(x, y, z)$, where $\varphi(x, y, z)$ can be any formula containing free variables x, y, and z, is equisatisfiable with $\forall x, y\ \varphi(x, y, f(x, y))$, where f is a fresh function symbol. This transformation can be repeatedly applied to any formula in prenex form in order to remove all existential quantifiers.

Definition 20 (Herbrand sets)**.** Given a many-sorted first-order language $\mathcal{L} = (\mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, r, d)$ and a formula φ in language $\mathcal{L}$, the family H of *Herbrand sets* for φ assigns for each sort $\tau \in S$, the smallest set H_τ such that:

- for each constant $a \in \mathcal{F}$ of sort τ used in φ, $a \in H_\tau$. If no constant of sort τ is used in φ, then H_τ includes one new arbitrary constant, so that H_τ is never empty;
- for each function $f \in \mathcal{F}$ of sort $(\tau_1, \dots, \tau_n, \tau)$ used in φ, and for all elements $t_1 \in H_{\tau_1}, \dots, t_n \in H_{\tau_n}$, the term $f(t_1, \dots, t_n)$ is in H_τ.

The Herbrand set H_τ is essentially the set of ground τ-terms than can be generated from the function symbols in φ, possibly with an additional constant symbol if none was present in φ.

Definition 21 (Congruence relation)**.** Given a set of terms T, a *congruence* relation R^C on T is a reflexive, symmetric and transitive relation such that if $f(t_1, \ldots, t_n), f(t'_1, \ldots, t'_n) \in T$ and if $(t_i, t'_i) \in R^C$ for each $i \in \{1, \ldots, n\}$, then $(f(t_1, \ldots, t_n), f(t'_1, \ldots, t'_n)) \in R^C$.

A congruence relation R^C on T partitions T into a set of *congruence classes*, also called *quotient set*, denoted T/R^C. A congruence class is a (maximal) subset of T in which all elements are congruent, i.e., for each pair of terms t, t' in a congruence class, $(t, t') \in R^C$. For more details and examples on congruence relations, see Section 6.1 on congruence closure. Each interpretation $\mathcal{I}$ defines a congruence relation $R^C_\mathcal{I}$ on the set of terms: $(t, t') \in R^C_\mathcal{I}$ if and only if $\mathcal{I} \models t = t'$.

Definition 22 (Herbrand interpretation)**.** Given a formula φ in a many-sorted first-order language and the associated family of Herbrand sets H, a *Herbrand interpretation* $\mathcal{I} = (D, I_\mathcal{V}, I_\mathcal{F}, I_\mathcal{P})$ is an interpretation such that:

- there exists a congruence relation R^C_τ such that D assigns H_τ/R^C_τ to sort τ;
- for any constant a of sort τ in φ, $\mathcal{I}[a] = \mathcal{C}_a$, where $\mathcal{C}_a$ is the class in H_τ/R^C_τ containing a;
- for any term $f(t_1, \ldots, t_n)$, if $\mathcal{I}[t_i] = \mathcal{C}_{t'_i}$ for each $i \in \{1, \ldots, n\}$, then $\mathcal{I}[f(t_1, \ldots, t_n)] = \mathcal{C}_{f(t'_1, \ldots, t'_n)}$.

For any ground term t, and Herbrand interpretation $\mathcal{H}$, we have $\mathcal{H}[t] = \mathcal{C}_t$. A *Herbrand model* is a Herbrand interpretation which is also a model.

Theorem 13 (Herbrand theorem)**.** *A (set of) closed Skolem formula(s) is satisfiable if and only if it has a Herbrand model.*

Theorem 14 (Second Herbrand theorem)**.** *A (set of) closed Skolem formula(s) is unsatisfiable if and only if there is a finite unsatisfiable set of instances of this (these) formula(s).*

These last two results allow the elaboration of a semi-decision procedure to check the satisfiability of first-order formulas (Theorem 10). Here is a brief overview of how this is done. Formulas that contain free variables can easily be transformed into an equisatisfiable closed formula by replacing each

occurrence of a free variable by a corresponding (fresh) constant. Theorem 13 reduces the problem of finding a model among all possible interpretations to finding a model among Herbrand interpretations. This allows to reduce the satisfiability problem for sets of ground formulas to SAT, essentially by assigning a proposition to each atom (we skip over the details).

A set containing closed quantified formulas in Skolem has the same Herbrand models as the set of instances of those formulas on the corresponding Herbrand sets. This allows reducing the satisfiability problem of closed first-order Skolem formulas to the satisfiability of a (usually infinite, but always countable) set of propositional formulas. The compactness theorem (Theorem 8) guarantees that if this set is unsatisfiable, there exists a finite unsatisfiable subset. This leads to theorem 14 and to a semi-decision procedure by enumerating the finite subsets of propositional formulas in a fair order and evaluating their satisfiability. If the (set of) first-order formula(s) is unsatisfiable, one of those subsets will as well, and vice-versa, meaning unsatisfiability can be shown in finite time.

2.5 Satisfiability modulo theories

Herbrand theory leads to a generic semi-decision procedure, based on propositional SAT solving, for the satisfiability of first-order formulas. This is great, but turns out to be not very effective at dealing with certain theories (i.e., sets of formulas) which are very useful in practice. Examples of such theories include integer arithmetic, or the theory of arrays. Furthermore, such theories sometimes contain axiom schemas leading to infinitely many axioms. This means the theory cannot be easily communicated to the solver, as it cannot be represented as a finite set of formulas. On the other hand, many specialized and efficient techniques have been developed for reasoning on those theories, one example being the simplex method for linear arithmetic. Satisfiability modulo theories aims at reconciling general SAT solving for first-order formulas with specialized solving techniques for first-order theories.

2.5.1 Theory solver

Theories can be described as a (possibly infinite) set of first-order formulas, called axioms. In recent literature, theories are instead usually defined as the set of interpretations which make those axioms true.

Definition 23 (Theory)**.** A *theory* $\mathcal{T}$ in first-order language $\mathcal{L}$ is an arbitrary set of interpretations on $\mathcal{L}$. A formula φ in language $\mathcal{L}$ is said to be $\mathcal{T}$-

satisfiable if and only if some element of $\mathcal{T}$ is a model of φ. A set Γ of ground formulas in language $\mathcal{L}$ $\mathcal{T}$-*entails* a ground formula φ, written $\Gamma \models_{\mathcal{T}} \varphi$ if and only if every model of Γ in $\mathcal{T}$ satisfies φ as well.

Definition 24 (Theory solver)**.** A *theory solver* for a theory $\mathcal{T}$ ($\mathcal{T}$-*solver*) is a procedure which takes as input a collection of literals in the language $\mathcal{L}$ of $\mathcal{T}$ and decides whether it is $\mathcal{T}$-satisfiable.

2.5.2 Ground SMT solving

In SMT, decision procedure for the $\mathcal{T}$-satisfiability of ground formulas are based on an extension of CDCL (cf. Section 2.2) called CDCL($\mathcal{T}$), also often seen as DPLL($\mathcal{T}$) [21] (DPLL being an ancestor of CDCL). As for CDCL, the details are glossed over.

CDCL($\mathcal{T}$) works with two parts: a CDCL-based SAT solver, and a theory solver for theory $\mathcal{T}$. First, the SAT solver treats each literal in the formula as a proposition, and tries to find an assignment. When an assignment is found, one needs to check whether the truth value assigned to each literal is consistent with theory $\mathcal{T}$ (i.e., if it is $\mathcal{T}$-satisfiable). The $\mathcal{T}$-solver is called to check if this is the case. If yes, the assignments returned by the SAT solver and the $\mathcal{T}$-solver form a model, and the formula is satisfiable. Otherwise, the $\mathcal{T}$-solver needs to provide an explanation for why the set of literals in $\mathcal{T}$-unsatisfiable, in the form of a clause. Such an explanation can always be generated: in the worst case, the simple negation of the set of literals can be taken as explanation, but as for conflict clauses in CDCL, smaller explanations (i.e. smaller clauses) reduce the search space more, and are thus preferred if they can be (efficiently) generated by the $\mathcal{T}$-solver. Once this clause is added, the current assignment of the SAT solver becomes invalid, and the SAT solver continues the search for another assignment. If no boolean assigment of the literals can make the formula true, the formula is unsatisfiable.

As for CDCL, many additional techniques are employed to make this process more efficient in practice.

2.5.3 Theory combination

In many cases, multiple theories can be involved. When that happens, the theory solvers must work together to find a satisfying assignment common to all theories. Fortunately, there exist methods for reconciling multiple specialized theory solvers into a theory solver for the combinations of the theories. One such theory combination method is the Nelson-Oppen combination procedure [22].

Definition 25 (Disjoint languages)**.** Two first-order languages $\mathcal{L}_1 = (\mathcal{S}_1, \mathcal{V}_1, \mathcal{F}_1, \mathcal{P}_1, r_1, d_1)$ and $\mathcal{L}_2 = (\mathcal{S}_2, \mathcal{V}_2, \mathcal{F}_2, \mathcal{P}_2, r_2, d_2)$ are *disjoint* when $\mathcal{F}_1 \cap \mathcal{F}_2 = \mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$, i.e., when both languages have no predicate or function in common.

Definition 26 (Union of disjoint languages)**.** The *union* of two disjoint languages $\mathcal{L}_1 = (\mathcal{S}_1, \mathcal{V}_1, \mathcal{F}_1, \mathcal{P}_1, r_1, d_1)$ and $\mathcal{L}_2 = (\mathcal{S}_2, \mathcal{V}_2, \mathcal{F}_2, \mathcal{P}_2, r_2, d_2)$ is the language $\mathcal{L} = (\mathcal{S}_1 \cup \mathcal{S}_2, \mathcal{V}_1 \cup \mathcal{V}_2, \mathcal{F}_1 \cup \mathcal{F}_2, \mathcal{P}_1 \cup \mathcal{P}_2, r, d)$, where r is the function equal to $r1$ when its argument is in the domain of $r1$, otherwise it is equal to $r2$. Function d is defined similarly.

Definition 27 (Restriction of an interpretation)**.** Given an interpretation $\mathcal{I}$ in language $\mathcal{L}$, a *restriction* of $\mathcal{I}$ to a subset of $\mathcal{L}$ is the interpretation equal to $\mathcal{I}$ for every domain, function symbol, predicate symbol in the subset of $\mathcal{L}$.

Definition 28 (Union of theories)**.** The *union* $\mathcal{T}$ of theories $\mathcal{T}_1$ and $\mathcal{T}_2$ in disjoint languages $\mathcal{L}_1$ and $\mathcal{L}_2$ is the set of all interpretations $\mathcal{I}$ such that the restriction $\mathcal{I}_i$ of $\mathcal{I}$ to $\mathcal{L}_i$ is in $\mathcal{T}_i$, for $i \in \{1, 2\}$.

Definition 29 (Stably infinite theory)**.** A theory $\mathcal{T}$ for language $\mathcal{L}$ is *stably infinite* if for any $\mathcal{T}$-satisfiable formula φ in $\mathcal{L}$, there is a model of φ in $\mathcal{T}$ whose domain is infinite.

Definition 30 (Convex theory)**.** A theory $\mathcal{T}$ for language $\mathcal{L}$ is *convex* if for any finite conjunction Γ of equalities of $\mathcal{L}$, $\Gamma \models_{\mathcal{T}} \bigvee_{1 \leq i \leq n} x_i = y_i$ if and only if $\Gamma \models_{\mathcal{T}} x_i = y_i$ for some $i \in \{1, \ldots, n\}$.

Nelson-Oppen offers a way to combine theory solvers for stably infinite theories on disjoint languages by sharing $\mathcal{T}_i$-entailed (disjunctions of) equalities between the theory solvers. If theories are convex, entailed disjunctions of equalities do not need to be shared, since at least one of the equalities in the disjunction can be deduced and shared on its own. If additionally the theory solvers can efficiently generate such entailed equalities, an efficient procedure for the union of the theories is obtained.

When at least one of the theories is non-convex, however, complexity skyrockets, because the number of possible disjunctions of equalities is exponential in the number of possible equalities. More recent theory combination schemes, such as delayed theory combination [23] or model-based theory combination [24] attempt to mitigate this by performing a CDCL-like search on which equalities hold between the shared variables.

2.5.4 Quantifiers

So far, the methods described for satisfiability modulo theories only work for ground formulas. They also work really well in practice. When quantifiers are present, arriving to efficient procedures becomes a lot more difficult. The usual methods for dealing with quantifiers in SMT are based on instantiation. First, a formula is Skolemized, then ground instances are generated from the quantified formulas using various techniques, and fed to the ground SMT solver in the hopes of finding a conflict. If the ground instances are unsatisfiable, then the formula is unsatisfiable. Otherwise, no conclusion can be drawn, but the model found for the ground instance may be used to choose which instances to generate next. Some of the techniques used for instantiation are the following:

- trigger-based instantiation using E-matching [25] [26], where the instances are generated based on the terms found in the formula;
- model-based quantifier instantiation (MBQI)[27] [28], where the model generated by the ground solver is extended to a full model for the quantified formula. Instances are then generated which try to contradict this model. If this is not possible, an actual model was found and the formula is satisfiable;
- conflict-based instantiation [29] and congruence closure with free variables [30], which are efficient techniques to detect and generate instances that contradict the current state of the ground solver;
- enumerative instantiation [31] [32], based on Herbrand theory, which tries to enumerate instances in some exhaustive heuristic order.

Chapter 3

Sudoku in SMT

In order to study the behavior of SMT solvers on finite domains, we will need to rely on example problems. In this work, we will mainly focus on the well-known Sudoku problem, as it can easily be encoded as an SMT problem and naturally relies on a finite domain. It is also representative of the kind of problems that may occur in an industrial setting[1]. As a reminder, the Sudoku problem consists in a 9 by 9 grid which must be filled with numbers 1 through 9, in such a way that each of the nine rows, columns and 3 by 3 subgrids (also called blocks) contains each number once, and only once. Given a grid with some of the numbers already filled in, the goal is to find the (unique) filling of the grid that satisfies all of these constraints. There exist many variants of this problem, including Sudoku on larger grids, such as 16 by 16 or 25 by 25.

3.1 Encoding

To encode a Sudoku in SMT, we start by labeling each row and each column with a number between 1 and 9. Then, we define a function $A : \mathcal{D} \times \mathcal{D} \longrightarrow \mathcal{D}$, with $\mathcal{D} = \{1, 2, ..., 9\}$, which maps each position in the grid to the number contained in that cell. We can see that the finite domain $\mathcal{D}$ naturally appears both in the domain and the image of A. In practice, the easiest way to represent the elements of $\mathcal{D}$ is to use the built-in integer sort, but we need to keep in mind that only the numbers 1 through 9 are relevant when encoding the constraints. The constraints are the following:

- $1 \leq A(i, j) \leq 9$ with $i, j \in \{1, \ldots, 9\}$;

[1] See for instance `https://www.clearsy.com/en/research-and-development/blasst/`

- $A(i,j) \neq A(i,k)$ with $i,j,k \in \{1,\ldots,9\}$ and $j \neq k$ (rows);
- $A(i,j) \neq A(k,j)$ with $i,j,k \in \{1,\ldots,9\}$ and $i \neq k$ (columns);
- $A(i,j) \neq A(k,l)$ with $i,j,k,l \in \{1,\ldots,9\}$, $i \neq k$ or $j \neq l$ and cells (i,j) and (k,l) in the same block;
- $A(i,j) = k$ with $i,j,k \in \{1,\ldots,9\}$ and number k in cell (i,j).

3.1.1 Quantifier-free encoding

The easiest way to encode the Sudoku constraints is simply to encode all of the constraints exhaustively. Since the grid is finite, so is the number of constraints. This kind of encoding is not only easy to understand for humans, it is also easier to deal with for the SMT solver than more succinct encodings using quantifiers, as we will see in Chapter 4. Some examples of the constraints to encode are:

- $A(2,4) \neq A(2,7)$;
- $A(5,7) \neq A(9,7)$;
- $A(5,4) \neq A(6,6)$;
- $1 \leq A(8,3) \leq 9$;
- $A(1,1) = 3$

3.1.2 Propositional encoding

Uninterpreted functions and integers are not actually necessary to solve Sudokus. Sudokus can be encoded with simple propositional logic. First create $9 \cdot 9 \cdot 9 = 729$ propositions $p_{i,j,k}$, with $1 \leq i,j,k \leq 9$. Proposition $p_{i,j,k}$ represents whether cell (i,j) contains number k. The constraints can then be encoded as such:

- Each cell contains at least one number:
  $\bigwedge_{i=1}^{9} \bigwedge_{j=1}^{9} \left(\bigvee_{k=1}^{9} p_{i,j,k}\right)$;
- Each cell contains at most one number:
  $\bigwedge_{i=1}^{9} \bigwedge_{j=1}^{9} \bigwedge_{1 \leq k < l \leq 9} (\neg p_{i,j,k} \vee \neg p_{i,j,l})$;
- Each cell in the same row must be different:
  $\bigwedge_{i=1}^{9} \bigwedge_{1 \leq j < k \leq 9} \bigwedge_{l=1}^{9} (\neg p_{i,j,l} \vee \neg p_{i,k,l})$;

- Each cell in the same column must be different: $\bigwedge_{1 \leq i < j \leq 9} \bigwedge_{k=1}^{9} \bigwedge_{l=1}^{9} (\neg p_{i,k,l} \vee \neg p_{j,k,l})$;
- Each cell in the same block must be different. For each (unordered) pair of distinct cells (i, j) and (k, l) within a same block, add the constraint: $\bigwedge_{m=1}^{9} (\neg p_{i,j,m} \vee \neg p_{k,l,m})$;
- For each known number k in cell (i, j), the corresponding proposition must be true: $p_{i,j,k}$.

The obvious problem with this encoding is that it is very unintuitive compared to the others. It might well be the easiest to solve for the SMT solver (only the embedded SAT solver would be at work here), but the cost of distilling different problems down to a propositional encoding can be quite high. One of the purposes of using more expressive logics is to avoid this cost.

3.1.3 Encoding using quantifiers

Having to encode each constraint individually can be cumbersome, especially if we want to encode larger problems, such as a 16 by 16 or 25 by 25 Sudoku. We only really have four kinds of constraints that apply to the whole grid, so we can get away with only encoding four *quantified* constraints. One set of suitable quantified formulas (barring the constraints for known numbers, which still need to be encoded individually) is:

- $\forall i, j.\, 1 \leq i, j \leq 9 \Rightarrow 1 \leq A(i, j) \leq 9$;
- $\forall i, j, k.\, (1 \leq i, j, k \leq 9 \wedge j \neq k) \Rightarrow A(i, j) \neq A(i, k)$ (rows);
- $\forall i, j, k.\, (1 \leq i, j, k \leq 9 \wedge i \neq k) \Rightarrow A(i, j) \neq A(k, j)$ (columns);
- $\forall i, j, k, l.\, (1 \leq i, j, k, l \leq 9 \wedge \lfloor \frac{i-1}{3} \rfloor = \lfloor \frac{k-1}{3} \rfloor \wedge \lfloor \frac{j-1}{3} \rfloor = \lfloor \frac{l-1}{3} \rfloor \wedge (i \neq k \vee j \neq l)) \Rightarrow A(i, j) \neq A(k, l)$ (blocks);

where all quantifiers range over the integers. This kind of encoding of course generalizes better to other sizes, and does not require adding constraints when the size is increased. It also seems more natural to encode each Sudoku rule as a single constraint rather than as a constraint on each cell. Similarly, in an industrial setting, expressing constraints using quantifiers is more natural and helps confidence in the models and the verification task. However, this kind of encoding is much more difficult to handle from the solver's point of view. It generally leads to a much worse performance in terms of computation time, and sometimes the solver is unable to solve the problem at all.

Chapter 4

Performance of existing solvers

In this chapter, the performance of various state-of-the-art SMT solvers on finite domain problems is assessed using the Sudoku problem, given to the solvers using various encodings similar to those described in Section 3.1. The problems are encoded in the SMT-LIB language [3], a standardized interface to SMT solvers implemented by most state-of-the-art solvers. The solvers investigated are Z3 [33], CVC5 [34], and Yices [35]. Yices does not offer support for quantified formulas, so no measurement is given in those cases.

4.1 Satisfiable case

The first few tests consist in solving a 9 by 9 Sudoku using different encodings. In this case, there is a (unique) solution to the Sudoku, so the solver should return `sat` with a model of the formula. Results are in Table 4.1. The different encodings are the following:

1. The first encoding corresponds to a quantifier-free encoding *without* using the built-in integer sort. Instead, a sort is declared along with nine distinct elements of that sort corresponding to the Sudoku digits $\{1, \ldots, 9\}$. This avoids the need to use a theory solver for integer arithmetic. This also prevents the use of order predicates, such as $\leq$. Conditions like $1 \leq A(8,3) \leq 9$ can instead be encoded like: $A(8,3) = 1 \vee \ldots \vee A(8,3) = 9$. Other than that, the encoding is the same as in Section 3.1.1;

2. The second encoding corresponds to the quantifier-free encoding described in Section 3.1.1, using the built-in integer sort;

3. The third encoding is a quantified encoding corresponding to the one given in Section 3.1.3, with the slight difference that the block con-

straints are encoded using modulo operators instead of integer division. This is because integer division is not part of the SMT-LIB standard. Modulos are not part of it either, but are supported by all three solvers as part of their integer arithmetic facilities. Note that with this encoding, all the quantified constraints are limited to the 9 by 9 grid of the Sudoku, even if A is defined on $\mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z}$;

4. The fourth encoding is similar to the third, but the constraint imposing numbers between 1 and 9 is extended to the whole infinite grid on which A is defined, i.e., $\forall i, j. 1 \leq i, j \leq 9 \Rightarrow 1 \leq A(i, j) \leq 9$ becomes $\forall i, j. 1 \leq A(i, j) \leq 9$. This is obviously still satisfiable (just assign any number between 1 and 9 to the infinitely many cells outside the 9 by 9 grid), but it tests the boundaries of what each solver is capable of, since a model is now subjected to infinitely many constraints, and can thus be harder to find;

5. The fifth encoding is similar to the fourth, with the addition that the rows and columns constraints are also not restricted to the grid anymore. For the rows, the constraint becomes $\forall i, j, k. (1 \leq j, k \leq 9 \wedge j \neq k) \Rightarrow A(i, j) \neq A(i, k)$, meaning the elements $A(i, 1), \ldots, A(i, 9)$ must be distinct for *any* row $i \in \mathbb{Z}$. A similar transformation is applied to the column constraint. This is satisfiable, for example by repeating any row from the 9 by 9 grid infinitely many times outside the grid, and similarly for the columns. For example, having $A(i, j) = A(6, j)$ for all $i \in \mathbb{Z} \setminus \{1, \ldots, 9\}$ and $j \in \{1, \ldots, 9\}$ satisfies all the row constraints outside the grid if they are satisfied for row 6;

6. The last encoding corresponds to a quantified encoding repeating the Sudoku constraints every 9 cells everywhere on the infinite grid. This is satisfiable by repeating the solved 9 by 9 Sudoku infinitely many times in all directions, i.e., by setting $A(i, j) = A(k, l)$ if $i \equiv k \pmod 9$ and $j \equiv l \pmod 9$. Multiple variations of this encoding were tried, all having the same unique model. Since they all give similar results, they are grouped into one here.

As can be seen in Table 4.1, the presence of integer arithmetic leads to a slight slow down, particularly noticeable for CVC5. It appears strange that CVC5 performs so poorly compared to the other two solvers. Looking into the scripts used by CVC5 for SMT-COMP [36] does not reveal any options that significantly improve this.

Quantifiers, as expected, make the problem more difficult. The time taken by each solver is almost multiplied by 10. The A function, constrained over

Encoding	Z3	CVC5	Yices
quantifier-free, no integers	0.024	0.450	0.163
quantifier-free	0.051	2.367	0.190
quantified, finite grid constraints	0.528	18.837	/
quantified, infinite grid constraints	0.550	19.839	/
quantified, infinite row/column constraints	0.627	dnf (> 12h)	/
quantified, repeated Sudoku constraints	dnf	dnf	/

Table 4.1: Time taken (in seconds) to solve a 9 by 9 Sudoku using various encodings (satisfiable case).

the integers, requires an infinite model, i.e., one that assigns an integer to every grid cell, even if the constraints are only effective within the grid. This leads CVC5 to give up and return `unknown` at first for any of the quantified encodings. Enabling the `--mbqi` option (for model-based quantifier instantiation, cf. Section 2.5.4) solves this problem and allows the solver to find a model for two of the four quantified encodings. It appears like Z3 automatically makes use of MBQI, allowing it to find a model too. The models constructed by MBQI for A are based on imbricated if-then-else constructs, assigning individual numbers to some finite amount of grid cells, and eventually assigning the same number to every cell beyond a certain point. With such constructs, the periodic function required to satisfy the constraints of the last encoding can never be found. It then makes sense that none of the solvers were able to find the assignment satisfying the last encoding. It is however unclear why CVC5 with MBQI is unable to find the solution for the fifth encoding.

4.2 Unsatisfiable case

The next tests are about checking unsatisfiability. To do that, the (unique) solution of the Sudoku is negated in the encoding. The solver then needs to try to find another solution, and should return `unsat` when realising that there is none. The same encodings are tested. The results, in Table 4.2, are not very different. The main differences are that everything is a bit faster, CVC5 even being four times as fast in the quantified case, and that Z3 and CVC5 are able to find solutions for every encoding, including the last one. This makes sense, since there is no need to find a complicated infinite model anymore: only a contradiction needs to be found, and it can be found within the initial 9 by 9 grid. The additional constraints do not make this any harder: they only add possibilities for more contradiction (unlike in the

satisfiable case where they added more constraints on the models, making them harder to find). It is also interesting to note that the `--mbqi` option is no longer necessary for CVC5. Other instantiation methods (likely trigger-based) are enough to generate the ground instances that lead to a conflict.

Encoding	Z3	CVC5	Yices
quantifier-free, no integers	0.022	0.511	0.010
quantifier-free	0.041	1.868	0.021
quantified, finite grid constraints	0.413	4.668	/
quantified, infinite grid constraints	0.496	4.658	/
quantified, infinite row/column constraints	0.412	4.815	/
quantified, repeated Sudoku constraints	0.471	4.564	/

Table 4.2: Time taken (in seconds) to verify that a 9 by 9 Sudoku has no other solution, using various encodings (unsatisfiable case).

4.3 Larger Sudokus

One of the advantages of using quantified encodings is that they are easy to generalise to larger sizes. Here is a quick study on the impact of the Sudoku size on solve time for quantified encodings. Only the first quantified encoding is used, i.e., the one described in Section 3.1.3, only applying constraints to the 9 by 9 grid, because it is the most natural out of all the quantified encodings studied. Chapter 5 develops a method for transforming a quantified encoding with effectively finite constraints such as this one into a quantifier-free encoding. This allows an easy testing of quantifier-free encodings (with integers) as well. As results are similar for the satisfiable and unsatisfiable cases on a 9 by 9 Sudoku using those encodings, only the satisfiable case is studied. Results are given in Table 4.3.

Encoding	Z3	CVC5	Yices
quantifier-free 9x9	0.051	2.367	0.190
quantified 9x9	0.528	18.837	/
quantifier-free 16x16	0.335	48.211	0.812
quantified 16x16	12.426	dnf (> 12h)	/
quantifier-free 25x25	84.729	dnf	61.607
quantified 25x25	730.783	dnf	/

Table 4.3: Time taken (in seconds) to solve Sudokus of multiple sizes using quantified and quantifier-free encodings.

The results show that solving time increases rapidly with the Sudoku size. This was expected, as it is pretty much what happens even using SAT solvers. Quantifiers only contribute to make this worse. In the 16 by 16 case, they even prevent CVC5 from solving. For some reason, CVC5 performs much worse than the other two solvers, preventing it from solving the quantifier-free 25 by 25 case.

Chapter 5

A first approach: dealing with quantifiers

We saw in Chapter 4 that SMT solvers can be much more effective when they do not have to deal with quantifiers. This becomes interesting when we realise that, when dealing with fixed finite domains, quantifiers are actually not necessary. Indeed, if the domain is finite, a quantified formula only has a finite number of instances. If the domain is known a priori, the formula can thus be replaced by the (finite) set of its instances. There is of course a trade-off here: if the domain is large or if many variables are quantified, the set of instances can be way too large to be used practically. Here, we explore ideas to eagerly instantiate formulas quantified over finite domains as a way to improve the performance of existing solvers. We will focus on formulas that are in Skolem normal form, that is formulas for which all the quantifiers are at the front, ranging over the whole formula, and all quantifiers are universal quantifiers. A set of formulas can be converted to an equisatisfiable set of Skolem forms via a process known as Skolemization, which is often applied as a first step in solvers (cf. Section 2.4).

5.1 Detecting finite domains over integers

In the Sudoku example, we are using integers to represent elements of our finite domain. This can be very useful as a way to encode the problem, and might be used as a way to encode other similar problems that work on a finite domain. From the solver's perspective, however, this is problematic, as it hides the fact that the domain is finite. The domain of integers is indeed not finite, and the domain is only effectively finite because of the constraints that are expressed in the formulas. If we want to apply techniques to work

on finite domains, we first need a way to detect these finite domains.

One thing to keep in mind is that, whatever we might do with a supposed finite domain, the solver must still return the right answer for the formula it was given. If a formula is universally quantified over the integers, for example, finding a satisfying assignment still means finding an assignment that makes the formula true for *all* the integers, not only those in the finite domain. In that sense, the actual domain of the (set of) formula(s) is still infinite. The reason why we can say that there is an "effective" finite domain is because some finite part of the problem contains most of the difficulty. Solving only that part would make solving the whole problem on the infinite domain easy.

There is a great variety of ways to encode effectively finite problems on the integers (i.e. problems with a finite "difficult" part encoded over the integers), and detecting all of them automatically would be an extremely difficult task. In fact, this is undecidable since it is quite straightforward to make the problem of recognizing finite problems for some formulas as hard as satisfiability checking for full first-order logic. Instead, we can try to detect some common occurrences of finite problems, which are similar in their structure. This is what is attempted here.

Looking at the formulas of Section 3.1.3 for Sudokus, one can notice the following: when i or j is outside of $\{1, \ldots, 9\}$, each formula becomes automatically true, no matter the value of $A(i, j)$. In other words, integers that are not part of the domain make the formula trivially true when used to instantiate a variable. This can be used as a *definition* of the "effective" finite domain.

Definition 31 (Effective domain)**.** For a formula of the form $\forall x, y_1, \ldots, y_n \ \varphi(x, y_1, \ldots, y_n)$, where x is quantified over the integers and $\varphi(x, y_1, \ldots, y_n)$ is an arbitrary formula containing free variables $x, y_1, \ldots, y_n$, we define the *effective domain* for variable x as the set of integers d such that $\not\models \varphi(d, y_1, \ldots, y_n)$. If this set is finite, we call it the *effective finite domain* of variable x.

In other words, we can say that an integer is outside of the effective finite domain if and only if the instances of the formula using this integer are valid. This definition makes it possible to extract the finite "hard" part of the problem, and obtain the solution to the complete problem by solving this extracted problem, as shown in the rest of this chapter. This definition is also quite general: it can work with many more problems than just Sudokus in that specific encoding. All finite domain problems can be encoded over the integers in a similar way: map each element of the finite domain $\mathcal{D}$ to an integer, then convert each Skolem formula over $\mathcal{D}$ to a Skolem formula

over the integers by adding the condition that the quantified variables are integers that correspond to elements of $\mathcal{D}$. For example, if elements of $\mathcal{D}$ are mapped to integers 1 to $|\mathcal{D}|$, the formula

$$\forall x_1 \ldots \forall x_n.\, \varphi(x_1, \ldots, x_n)$$

where $\varphi(x_1, \ldots, x_n)$ is any quantifier-free formula, becomes

$$\forall x_1 \ldots \forall x_n.\, (1 \leq x_1 \wedge x_1 \leq |\mathcal{D}| \wedge \ldots \wedge 1 \leq x_n \wedge x_n \leq |\mathcal{D}|) \Rightarrow \varphi(x_1, \ldots, x_n)$$

The two problems are then equisatisfiable. Most natural encodings of finite domain problems over the integers will be similar to this, in such a way that they are also captured by the above definition. In the rest of this chapter, we may refer to the "effective finite domain" defined up here as just "the finite domain", or even "the domain", when the meaning is unambiguous.

Now, we need a way to identify the elements of the finite domain among the integers. We are going to assume that the finite domain is represented as a contiguous block of integers. This allows us to represent the domain as an interval between and including its first and last element, rather than a list of all its elements. If this is not the case, we can instead work with a larger finite domain, again the set of integers between the first and the last element of the initial finite domain (both included), even if some are unused. For example, if the finite domain is $\{1, 3, 6\}$, we can chose to work with $\{1, 2, 3, 4, 5, 6\}$ instead. This does not impact the correctness of the following approach, although it can influence performance, especially if the gaps between the elements of the domain are large. With this assumption, the problem reduces to finding the smallest and the largest elements of the domain.

Consider first a single formula quantified over one integer variable: $\forall x.\, \varphi(x)$, where x is the quantified variable and $\varphi(x)$ is a quantifier-free formula in which x is a free variable (extensions to this will be made in Section 5.3). We can check whether an integer n is greater than the largest element of the finite domain $\mathcal{D}$ ($n > \sup \mathcal{D}$) for variable x by checking the validity of the following formula: $\forall x.\, \varphi(x) \vee x < n$. Indeed, suppose $n > \sup \mathcal{D}$: if $x \notin \mathcal{D}$, then by the above definition of the finite domain, $\varphi(x)$ is valid, so it will always be true. If $x \in \mathcal{D}$, it should be smaller than n, and $x < n$ would be true. This proves that the formula is valid if n is greater than the largest element of the domain. The converse is also true: if $\varphi(x) \vee x < n$ is valid for all x, then $\varphi(x)$ has to be valid for all $x \geq n$. This means that any $x \geq n$ is outside the domain, and n is thus greater than every element in the domain.

Similarly, we can check whether n is smaller than the smallest element of the domain by checking the validity of $\forall x.\, \varphi(x) \vee x > n$. The validity of these

formulas could simply be checked by using the SMT solver itself, using the fact that a formula is valid if and only if its negation is unsatisfiable. Note that the formula fed to the solver would not actually be quantified: since we check the (un)satisfiability of the negation of a universally quantified formula, the universal quantifier would be turned into an existential one, and then removed by Skolemization. The formula that would actually be checked would be of the form $\neg\varphi(x) \wedge x \geq n$, where x is a free variable.

We can now check whether an integer n is greater (resp. smaller) than the largest (resp. smallest) element in the domain. This is in fact enough to find the extremities of the domain relatively easily, using a dichotomy method. Start by finding an upper bound for $\sup \mathcal{D}$ and a lower bound for $\inf \mathcal{D}$. For the upper bound on $\sup \mathcal{D}$, this can be done by doing a first check for $n = 1$ (or any other positive value), then doubling it until we find an n which is greater than $\sup \mathcal{D}$. This ensures that we find an upper bound in $O(\log |\sup \mathcal{D}|)$ checks, if this upper bound exists. The same can be done for the lower bound on $\inf \mathcal{D}$ by starting with a negative value. Once these bounds are found, we can refine them and find the exact values of $\sup \mathcal{D}$ and $\inf \mathcal{D}$ using dichotomy: if n is the upper bound we found for $\sup \mathcal{D}$, we know that $\sup \mathcal{D}$ is between $n/2$ and n (or between the lower bound for $\inf \mathcal{D}$ and n, if n is the starting value). The same can be done for $\inf \mathcal{D}$. Overall, this whole procedure for finding $\sup \mathcal{D}$ and $\inf \mathcal{D}$ can be done in $O(\log |\sup \mathcal{D}| + \log |\inf \mathcal{D}|)$ checks.

This procedure assumes that an effective finite domain can be found within the problem: if the SMT problem over the integers does not encode a finite domain problem, or if the encoding is too complicated to fit the definition of the "effective finite domain" above, then there is no effective finite domain to be found. In that case, the procedure may not terminate. If it is not known whether an effective finite domain exists, we can simply add a maximum in the search for $\sup \mathcal{D}$ (or a minimum for $\inf \mathcal{D}$). If an upper bound is not found below this maximum, simply stop the search and assume the effective domain is infinite. In practice, encountering a problem for which $\sup \mathcal{D}$ is above a value like 2^{64} is rather unlikely, and searching up to that value means the procedure would terminate in around $4 \cdot 64 = 256$ checks in the worst case, which is reasonable. Moreover, if the domain is very large, the techniques we develop here in Section 5.2 would be too slow to be practical. Not detecting such large domains is therefore a minor issue.

5.2 Removing quantifiers with exhaustive instantiation

Once the finite domain is identified, the hard part is over, as getting rid of the quantifiers is fairly straightforward. Since a formula is always valid outside the domain, all there is to worry about is the finite domain itself. It suffices to replace the formula by the conjunction of its instances on the finite domain. The resulting formula is logically equivalent to the original formula.

This is fairly easy to prove: every interpretation of a formula universally quantified over the integers assigns the same truth value to the formula and to the set of its instances, i.e, the formula has the same models as its set of instances. Furthermore, since the all the instances outside the effective finite domain are valid (by definition of the effective domain), they are true in every interpretation and do not impact the truth value assigned to the set. This means they can be removed from the set of instances without changing the models of the set. The set of instances restricted to the effective finite domain thus has the same models as the quantified formula. Naturally, the conjunction of those instance also does. Since the formula and the conjunction of its instances on the finite domain have the same models, they are logically equivalent.

The number of instances is exactly the size of the effective finite domain for the variable considered. This may cause problems if the size of the domain is very large. In that case, one may simply give up and revert to giving the original quantified formula to the solver.

5.3 Extension to multiple formulas quantified over multiple variables

In most cases, there is more than a single formula, and each formula is quantified over multiple variables. The above procedure can be extended to work on any finite set of Skolem formulas.

5.3.1 Multiple formulas

The first thing to notice is that each formula can have a different finite domain (according to the way we detect finite domains): for example, one formula may be valid when instantiated with a value outside $\mathcal{D}_1 = \{1, 2, 3\}$, a second formula may only be valid when instantiated with values outside

$\mathcal{D}_2 = \{1, 2, 3, 4\}$, and a third formula may not even have valid instances at all ($\mathcal{D}_3 = \mathbb{Z}$). In that case, there is no global finite domain, as the third formula's domain is not finite, but work can still be done to remove quantifiers on the first two formulas. If we have a procedure that can detect finite domains and remove quantifiers from single formulas, this procedure can simply be applied to each formula individually. If a "global" finite domain $\mathcal{D}$ exists (one that is the same for every formula), this procedure is guaranteed to find formula specific domains $\mathcal{D}_i$ which are either the same as $\mathcal{D}$ or included in $\mathcal{D}$ ($\mathcal{D}_i \subseteq \mathcal{D}$). This is because the existence of a global domain $\mathcal{D}$ would imply that if any element $e \in \mathbb{Z} \setminus \mathcal{D}$ is used to instantiate *any* formula, this formula would become valid. This is precisely how the formula specific domain $\mathcal{D}_i$ would be detected: by finding all elements which make the formula valid when instantiating it. We would find that e is one of those elements, and thus $e \notin \mathcal{D}_i$. If an element of $\mathcal{D}$ makes a formula valid when used to instantiate it, then that element would not be detected as being part of $\mathcal{D}_i$ (and thus $\mathcal{D}_i \subset \mathcal{D}$). This is not a problem as it would only reduce the number of instances considered when removing the quantifiers of this formula. It would result in a shorter but equivalent "dequantified" formula.

5.3.2 Formulas quantified over multiple variables

Now, when it comes to formulas that have multiple quantifiers, the same idea can be applied: a finite domain does not have to be detected for the whole formula, finite domains can be identified on a per quantifier basis. This is also useful when dealing with formulas quantified over different sorts (not only integers): while this procedure cannot deal with quantifiers that range over, say, the real numbers, it might still be possible to remove some of the quantifiers that range over the integers within the same formula.

Consider a Skolem formula $\forall x_1 \ldots \forall x_n.\, \varphi(x_1, \ldots, x_n)$ (where $\varphi(x_1, \ldots, x_n)$ is quantifier free). A procedure to (try to) eliminate a quantifier from the formula can be built as follow:

1. Push the quantifier to eliminate to the far right:
   $\forall x_1 \ldots \forall x_{i-1} \forall x_{i+1} \ldots \forall x_n \forall x_i.\, \varphi(x_1, \ldots, x_n)$.
   This is always possible since top level universal quantifiers permute. This step is not strictly necessary, but it avoids feeding quantified formulas to the solver when checking for bounds, which is really important in practice.

2. Consider only subformula $\forall x_i.\, \varphi(x_1, \ldots, x_n)$ (where $x_1, \ldots, x_{i-1}$ and $x_{i+1}, \ldots, x_n$ are all free variables) and try to remove the quantifier as above in the case of a single quantifier. For example, if the subformula

is found to have an effective finite domain $\mathcal{D}_i = \{1, 2\}$, then it is equivalent to $\varphi(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) \wedge \varphi(x_1, \ldots, x_{i-1}, 2, x_{i+1}, \ldots, x_n)$.

3. Since the resulting formula is logically equivalent to the original subformula, it can simply be replaced inside the complete Skolem formula, which now has one less quantifier.

This process can simply be repeated for all the quantifiers that range over the integers.

Similarly to the last section, it can be shown that if there exists an effective finite domain $\mathcal{D}$ for the whole formula, then the finite domains $\mathcal{D}_i$ found for each quantifier are equal to or included in $\mathcal{D}$. Indeed, for any element $e \in \mathbb{Z} \setminus \mathcal{D}$, we have that $\varphi(x_1, \ldots, x_{i-1}, e, x_{i+1}, \ldots, x_n)$ is valid, no matter the values of i, $x_1, \ldots, x_{i-1}$ and $x_{i+1}, \ldots, x_n$. When considering the subformula $\forall x_i.\, \varphi(x_1, \ldots, x_n)$, we would thus find that $e \notin \mathcal{D}_i$, since the formula is valid when instantiated with that value. Since no element outside of $\mathcal{D}$ can be in $\mathcal{D}_i$, we have $\mathcal{D}_i \subseteq \mathcal{D}$.

As a result, if all the quantifiers range over the integers and the formula has a finite domain $\mathcal{D}$, the resulting formula will be a quantifier free formula of size at most $|\mathcal{D}|^n$ times the size of $\varphi(x_1, \ldots, x_n)$ (assuming the finite domain is a contiguous block of integers, cf. Section 5.1). If some quantifiers do not have a finite domain or range on sorts other than the integers, the other quantifiers can still be removed by this process.

5.4 Implementation and results

A prototype implementation of this dequantification procedure was made using PySMT [37], a python API for SMT, which supports the SMT-LIB format [3]. This implementation was then tested against the quantified encoding from Section 3.1.3. Some tweaking of the PySMT parser was necessary before it could handle the modulo operator used for that encoding, as modulo is not standard in SMT-LIB.

PySMT does not provide an easy way to Skolemize formulas. An alternative way of doing things is to use the fact that dequantified formulas are logically equivalent to the starting quantified formula. An algorithm for dequantifying a formula of any form is the following:

1. Transform existential quantifiers into universal quantifiers by negating them: $\exists x(\varphi(x)) \longleftrightarrow \neg\forall x(\neg\varphi(x))$.

2. Starting from the bottom of the syntax tree, (try to) transform subformulas of the form $\forall x\ \varphi(x)$ into a logically equivalent formula ψ

where the top level quantifier has been removed, using the procedure described in the previous sections.

Note that step one can be performed at the same time as step two while walking through the syntax tree. This approach has one drawback compared to Skolemizing first: if some sub-formula cannot be dequantified (be it because the quantifier does not range over the integers or because there is no effective finite domain), a (sub-)formula containing it cannot be dequantified without evaluating a quantified formula. Indeed, Skolemizing allowed the quantifiers to be swapped such that the formula for checking bounds was always a quantifier-free formula, but this would not be the case anymore. As an example, take a formula $\forall x. \neg\forall y\varphi(x, y)$, where $\varphi(x, y)$ is quantifier-free, and the quantifier $\forall y$ cannot be removed because it ranges over the reals. Skolemizing gives the formula $\forall x. \neg\varphi(x, f(x))$. Checking whether the formula is valid when $x \geq n$ amount to checking whether $x \geq n \wedge \varphi(x, f(x))$ is unsatisfiable. This last formula is quantifier-free, so this is relatively easy. If we had used the above procedure without Skolemizing, checking the same bound would amount to checking whether $x \geq n \wedge \forall y\varphi(x, y)$, which can be much more difficult because of the quantifier. Because the quantified encoding of Sudoku on which we test this procedure is already in Skolem form, and this is only a prototype implementation, this is not an issue in this case.

The quantified encoding given in Section 3.1.3 has an effective finite domain $\mathcal{D} = \{1, \ldots, 9\}$ for each quantifier, meaning they can all be removed using this procedure. The times taken to remove the quantifiers for different sizes are given in Table 5.1.

Encoding	Z3	CVC5	Yices
9x9	4.068	8.180	4.913
16x16	15.680	20.028	16.254
25x25	73.546	78.394	74.249

Table 5.1: Time taken (in seconds) to remove quantifiers from the quantified encoding of Sudokus of multiple sizes, using the procedure developed in Chapter 5.

It turns out that dequantifying takes a fair bit of time even for small Sudokus, likely because of the numerous calls to the SMT solver used for checking bounds. However, the time taken does not explode when dequantifying larger Sudokus like it does when solving them. In the quantifier-free encoding, the number of constraints to encode grows cubically with respect to the size of the grid. It appears like that could fit the times obtained here. For large Sudokus, dequantifying before solving could thus save a fair bit of

time compared to solving the quantified Sudoku directly. This comparison is made in Table 5.2.

Encoding	Z3	CVC5	Yices
dequantify first 9x9	4.119	10.547	5.103
solve quantified 9x9	0.528	18.837	/
dequantify first 16x16	16.015	68.239	17.066
solve quantified 16x16	12.426	dnf (> 12h)	/
dequantify first 25x25	158.275	dnf	135.856
solve quantified 25x25	730.783	dnf	/

Table 5.2: Total time taken (in seconds) to solve Sudokus either directly in their quantified encoding, or by dequantifying first.

For small Sudokus, dequantifying takes a significant amount of time and is not worth it for Z3. For CVC5, however, it is already twice as fast for the 9 by 9. When the Sudokus become larger, the advantage becomes much more significant. On the 25 by 25 Sudoku, it shaves nearly 10 minutes off of the solving time for z3. This is an almost 80% reduction. CVC5 is not able to solve even the dequantified version of the 25 by 25, but dequantifying allows it to solve the 16 by 16 in only a minute, which it could not solve in the quantified encoding. This dequantification procedure does not use any quantifier reasoning inside the SMT solver, allowing Yices to solve the Sudokus as well, although it has no quantifier capabilities. The `--mbqi` option is also no longer necessary for CVC5 in the dequantified case.

Chapter 6

Theory of uninterpreted functions over finite domains

The last chapter was a rather hands on, experimental approach to improving the performance of SMT solvers on finite domains. Here, the focus is on a more theoretical aspect. In SMT solvers, congruence closure is the main algorithm used to reason about the theory of equality with uninterpreted functions. On an infinite domain, it can be used to efficiently compute all the entailed equalities (that can then be propagated to the other theory solvers), and detect conflicts with disequalities. When the domain is finite, however, some constraints can arise due to the limited number of available elements. Those constraints are not taken into account by classical congruence closure. Usually, the conflicts are detected at a later stage when the different theory solvers fail to find a common satisfying assignment. Then, the underlying SAT solver is called to propose a new configuration, if there is one. This process can be rather inefficient. The goal of this chapter is to study ways to do this reasoning more efficiently by taking into account the domain cardinality constraints inside the theory solver itself.

6.1 Classical congruence closure

An equivalence relation is a binary relation on a set that has the property of being reflexive, symmetric, and transitive. Equality is an equivalence relation:

- Reflexivity: $\forall x.\, x = x$
- Symmetry: $\forall x, y.\, x = y \Rightarrow y = x$
- Transitivity: $\forall x, y, z.\, (x = y \wedge y = z) \Rightarrow x = z$

Subsets of elements which are all equivalent (and to which no other element can be added) form an equivalence class. A set equipped with an equivalence relation can always be partitioned into a set of equivalence classes, called the quotient set.

A congruence relation is an equivalence relation which preserves equivalence under some operations. In the case of the theory of equality with uninterpreted functions, we have an equivalence relation, equality, which is preserved under function application (monotonicity):

$$\forall x_1, \ldots, x_n, y_1, \ldots, y_n. (x_1 = y_1 \wedge \ldots \wedge x_n = y_n) \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$$

It is thus also a congruence relation.

The equivalence closure of a binary relation is the smallest equivalence relation which includes this relation. Similarly, its congruence closure is the smallest congruence relation which includes it. For example, given the set $S = \{a, b, c, f(b), f(c)\}$ and the binary relation $R = \{(a, b), (a, c)\}$ on S, we can compute its equivalence closure:

$$R^E = \{(a, a), (b, b), (c, c), (f(b), f(b)), (f(c), f(c)), (a, b), (b, a), (a, c), (c, a), (b, c), (c, b)\}$$

which leads to the quotient set $S/R^E = \{\{a, b, c\}, \{f(b)\}, \{f(c)\}\}$. Indeed, this is the smallest relation which contains R and is also reflexive, symmetric, and transitive. We can also compute the congruence closure:

$$R^C = R^E \cup \{(f(b), f(c)), (f(c), f(b))\}$$

Note that we will always have $R^E \subseteq R^C$. The equivalence classes (also called congruence classes) now are $S/R^C = \{\{a, b, c\}, \{f(b), f(c)\}\}$.

Computing the congruence closure of a relation gives us a way of easily checking the satisfiability of a quantifier-free formula in the theory of equality with uninterpreted functions. Consider the following formula:

$$a = b \wedge a = c \wedge f(b) \neq f(c)$$

From the formula, start by building the set of terms $S = \{a, b, c, f(b), f(c)\}$. Now, consider all equalities in the formula and build a relation over S from them: $a = b$ gives (a, b) and $a = c$ gives (a, c), we end up with the example from earlier: $R = \{(a, b), (a, c)\}$. Then, compute R^C, the congruence closure of R. R^C is the smallest set which is a congruence relation on S, and contains (a, b) and (a, c). In other words, if we interpret R^C as the equality

relation, it is the smallest possible equality relation for which $a = b$ and $a = c$. This means R^C contains all the equalities implied by the formula, and only the equalities that are implied. Now, to check if the formula is satisfiable, simply check whether the disequalities of the formula are in conflict with the equalities of R^C: we have $f(b) \neq f(c)$ in our formula, but we also have $(f(b), f(c)) \in R^C$, which means $f(b) = f(c)$ is implied by the rest of the formula. This formula is thus unsatisfiable. If there were no conflicts, the formula would be satisfiable: simply assign a different value to each congruence class in S/R^C. This would satisfy all the entailed equalities, and thus all the equalities of the formula. Since no two congruence classes are given the same value, this would also mean that every pair of elements that *can* be different *are* different. Since there are no conflicts, the disequalities would thus also be satisfied.

The computation of the congruence closure of a relation can be done efficiently ($O(n \log n)$, where n is the total number of symbols in the input equalities) [38]. In practice, there is no need to keep track of R^C as set of pairs of elements which are equal (this could have a quadratic size), we can simply keep track of the set of congruence classes. This is done using a union-find data structure, so that congruence classes can be merged efficiently when two elements within them are found to be equal.

It is also useful to represent the dependencies of the terms with one another. For example, $f(a, b)$ depends on a and b, $f(f(a, b))$ depends on $f(a, b)$,... These dependencies are commonly represented as a directed acyclic graph, where the vertices are the different terms, and an edge between term a and term b means that term a is directly dependant on b. The construction of such graphs is fairly straightforward. They are mostly useful to keep track of the direct predecessors of each term in the graph, i.e. the terms which are directly dependant on a given term. This is important because when two congruence classes are merged, we want to know which terms are dependant on the terms that were merged, in order to be able to keep the congruence rule consistent. Say we have the following terms in their congruence classes: $\{\{a\}, \{b\}, \{f(a)\}, \{f(b)\}\}$. After handling the equality $a = b$, the congruence classes of a and b are merged. It is important to notice that $f(a)$ and $f(b)$ depend on a and b, because by the congruence rule, their congruence classes now need to be merged too. Using this, the congruence closure can be computed in the following way. This is a simplified algorithm adapted from [39] for the sake of illustration. In practice, congruence closure needs to be backtrackable, incremental, be able to efficiently generate small conflict sets. This makes the overall algorithm quite a bit more complex. A description of such an algorithm can be found here: [38].

1. Start with each element of the set of terms S in its own congruence class.

2. For each equality in the input, merge the two congruence classes containing the two members of the equality. Note that if we were computing equivalence closure, we would be done after this step, but now we also need to make the classes consistent with the congruence rule.

3. Build a set A of the terms that need to be checked against the congruence rule. Initially, all the terms of S are in A.

4. For each term t in A, check against the terms that were previously checked (i.e. those in $S \setminus A$) if they are congruent due to the congruence rule. If a match is found, add their congruence classes to a list of congruence classes that need to be merged. Then, remove t from A.

5. For each pair of congruence classes that need to be merged, merge the smaller one into the bigger one. Add all the (direct) predecessors of the terms that changed congruence class (i.e. those that were in the smaller congruence class) back into A.

6. If A is empty, we are done, otherwise go back to step 4.

The check in step 4 can be done in constant time using a hash table. The hash table would map a tuple composed of a function symbol and an congruence class for each argument of the function to the congruence class corresponding to the result of the function given arguments in those congruence classes. Mathematically, this table can be written as $\tau : \bigcup_{n=0}^{\infty}(\Sigma^{(n)} \times C^n) \to C$, where $\Sigma^{(n)}$ is the set of function symbols of arity n found in S, and C is the current set of congruence classes. It can be thought of as a way to map a set of terms that are all congruent due to the congruence rule to a single congruence class.

If all the terms in $S \setminus A$ were previously encoded in such a hash table, it is easy to check if t is congruent to any of them: just check if the entry that would correspond to t in the table is already populated. There are three cases:

- If nothing is returned, none of the terms in $S \setminus A$ is congruent to t due to the congruence rule. Note that they might still be in the same of congruence class, or turn out to be congruent later when other congruence classes are merged.

- If the class returned is the congruence class of t, it means the congruence rule applies between t and some other element in $S \setminus A$, but they are already in the same congruence class, so there is nothing to be done.
- If the class returned is different from the congruence class of t, then there is some element in $S \setminus A$ that should be congruent to t but is currently in another congruence class. The two congruence classes will thus need to be merged.

6.2 Finite domain implications

Classical congruence closure computes, from a given set of equalities, all the entailed equalities. It does nothing with disequalities apart from checking if they are in conflict with the entailed equalities. This does not mean disequalities cannot be propagated: for example, given the disequality $f(a) \neq f(b)$, we can infer, by the contrapositive of the congruence rule, that $a \neq b$. Another example: given $a = b$ and $a \neq c$, we know that we necessarily have $b \neq c$.

Reasoning about equalities is usually enough: in the classical Nelson-Oppen procedure for theory combination [22], only the equalities need to be shared between the different theory solvers, and individual solvers are only responsible for checking if some formula within their own theory is satisfiable or not. With an infinite domain, finding all the entailed equalities is enough to decide satisfiability in the theory of equality with uninterpreted functions: if there are no conflicts between the entailed equalities and the given disequalities, a satisfying assignment can always be found (cf. last section), otherwise the formula is unsatisfiable.

When the domain is finite, however, this is not necessarily true anymore. Take for example the formula $a \neq b \wedge a \neq c \wedge b \neq c$ on a domain of size 2. The congruence closure algorithm would return the congruence classes $\{\{a\}, \{b\}, \{c\}\}$. Since there are no conflicts with the disequalities, the formula would be considered satisfiable. This would be true if the domain had at least 3 elements in it, such that we can assign a different element to each congruence class. With a domain of size 2, however, this is not possible, so we may not necessarily find a satisfying assignment. In this case, it is in fact not possible: a first element can be assigned to a, the second element needs to be assigned to b because of the first disequality, then c cannot be assigned anything since it must be different from both a and b, but there are no elements left in the domain. No satisfying assignment can be found. That said, the domain being too small does not necessarily lead to unsatisfiability:

if the formula had been $a \neq b \wedge a \neq c$, then the same set of 3 congruence classes would be reached by congruence closure, but a satisfying assignment could be found with $b = c$.

All of this happens because the domain size implies a hidden condition, that could be expressed as a disjunction of equalities: $a = b \vee a = c \vee b = c$. The problem is that this is a disjunction, something that the congruence closure algorithm is not built to handle. Since none of the equalities of the disjunction are known to be true on their own, the disjunction makes propagating equalities impossible, so reasoning about both equalities and disequalities is now necessary to decide satisfiability. This makes the problem much harder, and even NP-complete (cf. Section 6.4), so there is no hope of finding a polynomial time algorithm unless P = NP.

In a classical SMT solver, this kind of formula would be dealt with at the level of the SAT solver, which would try to guess which of the equalities hold and which ones do not, then call the congruence closure solver to verify if this assignment is possible. If not, it would keep trying other assignments until it runs out of possibilities, at which point the SMT solver is forced to return `unsat`. This process can be rather inefficient, as the SAT solver has no way to know in advance which assignments will be satisfiable (no reasoning is done on equality at all, it is just treated as an arbitrary predicate, which could always be true or false). If we could do the entire reasoning about the finite size of the domain inside the congruence closure solver, this back and forth would be avoided, and the problem would be solved at once by the theory solver. This is what is attempted in the rest of this work.

6.3 Application to Sudoku

Congruence closure over finite domains would apply very well to Sudokus. In the quantifier-free encoding of Sudokus (cf. Section 3.1), all the constraints are simply disequalities between elements of the same row, column, or block, or equalities with known elements of the domain. The theory of equality with uninterpreted functions is actually the only one involved. (In Section 3.1, integers are also involved, but they are only there to represent elements of the domain, they are not strictly necessary.) The only reason the congruence closure solver is unable to solve the Sudoku on its own is because the domain is of size 9, instead of infinite. Correctly filling a 9 by 9 Sudoku grid with infinitely many numbers to chose from would be really easy. If instead the congruence closure solver could take into account the finite domain constraint, the Sudoku would be entirely solved. If it could do that efficiently (or as efficiently as we can get for NP-complete problems), we would have an

efficient way to solve the Sudoku using the SMT solver.

6.4 NP-completeness

Congruence closure over finite domains can be shown to be NP-complete. The proof comes in two steps, as is always the case when proving NP-completeness. First, we prove that the problem is part of NP by showing that a solution can be generated non-deterministically and checked in polynomial time. Then, we show that the problem is NP-hard by polynomially reducing a known NP-complete problem to it (in this case, graph coloring).

The first part is fairly easy: given a formula in the theory of uninterpreted functions, a domain size $|\mathcal{D}|$, and a non-deterministically generated partition C of the terms appearing in the formula, representing the set of congruence classes, checking whether this partition satisfies the formula, the domain size constraint, and the congruence rule can be done in polynomial time. Equivalence axioms such as reflexivity, symmetry and transitivity are automatically satisfied by the properties of a quotient set.

If the formula has size n, where n is the number of (potentially nullary) function symbols appearing in the formula, the number of distinct terms is bounded by n. A table mapping each term to its congruence class in C can be built in linear time to allow subsequent constant time access. Then, checking whether the formula is satisfied is really easy: for each equality, check that the two terms are in the same congruence class; for each disequality, check that they are in different congruence classes. This is done in linear time with appropriate data structures. Checking the domain size constraint amounts to checking that the number of equivalence classes (i.e. $|C|$) is smaller than or equal to $|\mathcal{D}|$. This ensures that each equivalence class can be assigned a distinct element of the domain. This takes constant time. Finally, checking that the congruence rule holds can be done by running the congruence closure algorithm, starting with C as the set of congruence classes instead of having each term in a separate class. If any classes are merged, this means the congruence rule was not satisfied. If C is left unchanged, it was satisfied. Congruence closure has complexity $O(n \log n)$ [38], so this can be done in polynomial time too. This is assuming $O(1)$ complexity for hash tables, which is not actually the case in the worst case, but it stays polynomial. Note that if we use the algorithm from Section 6.1, we only need to go through the main loop once to know whether classes will be merged or not, so the overall complexity might be even lower.

To prove that the problem is NP-hard, we can show that there is a polynomial reduction from the graph coloring problem to congruence closure over

finite domains. Indeed, if it was possible to decide the satisfiability of a formula of the theory of uninterpreted functions under the domain cardinality constraint $|\mathcal{D}| \leq k$ for some positive integer k in polynomial time, then deciding whether an arbitrary graph is k-colorable (i.e., whether it is possible to assign one of k colors to each vertex such that no two adjacent vertices have the same color) would be doable in polynomial time too.

The reduction is done as follows: given a (undirected) graph $G = (V, E)$ with n vertices labelled $v_1, \ldots, v_n$, create fresh nullary symbols $t_1, \ldots, t_n$ corresponding to each vertex. Then, for each edge $\{v_i, v_j\} \in E$, add the disequality $t_i \neq t_j$ to the formula. The resulting formula is a conjunction of disequalities corresponding to each edge of the graph. It is easy to see that under the domain cardinality constraint $|\mathcal{D}| \leq k$, this formula is satisfiable if and only if G is k-colorable. Simply create a bijective map between the elements of $\mathcal{D}$ and the k colors. If the graph is k-colorable, then a satisfying assignment of the formula is obtained by assigning to t_i the element of $\mathcal{D}$ corresponding to the color of v_i. Conversely, if there is a satisfying assignment of the formula, a k-coloring of the graph is obtained by coloring v_i with the color that corresponds to the element assigned to t_i.

This finishes the proof that congruence closure over finite domains is NP-complete. One can also note that functions are not involved in the reduction, so we have an even stronger result: deciding the satisfiability of a formula in the theory of equality under a domain cardinality constraint $|\mathcal{D}| \leq k$ (with $k > 2$, as graph coloring is only NP-complete for $k > 2$) is NP-complete.

6.5 Approaching the problem using SAT

When confronted with an NP-complete problem, a common approach is to reduce the problem to a satisfiability problem in propositional logic, then use a state-of-the-art SAT solver to do the heavy lifting. Since SAT is NP-complete, it is possible to find a polynomial-sized encoding of any NP problem in SAT. The hope is then that the SAT solver is able to solve the problem efficiently in most cases. Obviously, due to NP-completeness, there are always some cases that will lead to exponential time. The effectiveness of this approach is dependent on the initial NP-complete problem, as well as the particular encoding that is used. Indeed, a bad encoding can lead to the SAT solver not being able to reason efficiently about the problem, even if the original problem was "simple".

As it turns out, two of the problems for which this approach is particularly effective, and often used in practice, are the Sudoku and graph coloring problems. This is promising because these two problems seem closely related to

the problem at hand. The link between the theory of uninterpreted function over finite domains and graph coloring is particularly apparent through the reduction in Section 6.4. Sadly, the reduction goes in the wrong direction: graph coloring can easily be encoded as a formula in the theory of uninterpreted functions restricted to a finite domain, but the converse is more complicated. If there was a "straightforward" reduction to graph coloring (i.e. a reduction where the resulting graph is not too complicated, or stays similar in structure to the original problem), the reduced problem could simply be encoded in SAT the way graph coloring is usually encoded, and this would hopefully lead to an efficient procedure for this problem.

In the case where only a single sort is involved in the formula, an almost straightfoward reduction to graph coloring is possible, after applying the classical congruence closure algorithm. Some additional constraints need to be encoded, but they can be added at the level of the SAT encoding. This method is explained in the next section. When multiple sorts are involved, possibly with different cardinality constraints for each sort, additional techniques are necessary. In particular, it can be shown that reasoning about each sort independently is not possible, as complex interactions can arise between elements of different sorts. The problem thus needs to be dealt with as a whole, rather than by trying to isolate the different parts. Different approaches for doing that are discussed in the following sections.

6.6 Algorithm for a single sort

In the case of a single sort, i.e. when all elements/symbols/variables are part of the same (finite) domain, the problem of satisfiability in the theory of uninterpreted functions over a finite domain almost reduces to graph coloring after applying congruence closure. Without the congruence rule, the problem would directly encode as a graph coloring problem, but a bit more work is necessary to take the congruence rule into account.

The first step is to apply congruence closure in the same way it would be applied if there were no cardinality constraints. All the equalities derived from reflexivity, symmetry, transitivity, or the congruence rule apply just as well in the case of a finite domain as they do for an infinite domain. The algorithm results in a set of congruence classes which is such that each pair of elements within a same class *need* to be equal according to the equalities given in the input formula, equality axioms, and the congruence rule. Each pair of elements which are in distinct classes *can* be different according to those rules. Nothing prevents two elements from two different classes from being equal (if they are equal, the corresponding congruence classes should be

merged, and the congruence closure should be recomputed). The congruence closure algorithm only merges classes that *must* be merged.

At this point, one can already check whether there is a conflict between the input disequalities and the resulting congruence classes. If two elements within the same class are supposed to be distinct, the formula is unsatisfiable regardless of the size of the domain.

Then, one can compare the domain size constraint with the resulting number of congruence classes. If the size of the domain is larger or equal to the number of congruence classes, then the problem is not different from when the domain is infinite: a satisfying assignment is obtained by assigning a different element to each congruence class. In this way, there can be no additional conflict with the input disequalities. If the domain is smaller than the number of congruence classes, this is not possible anymore, as there are not enough elements to make all the congruence classes different. We now need to selectively merge classes in such a way that the input disequalities and the congruence rule stay respected, until there are at most as many classes as elements in the domain. This is where the problem becomes much harder.

If we disregard the congruence rule, the remaining problem is actually equivalent to graph coloring: we want to know if we can assign $|\mathcal{D}|$ elements (or colors) to $|C|$ classes (or vertices) in such a way that the same element is not assigned to two classes which must be different (or the same color is not assigned to two vertices which are connected). This is easily encoded in SAT (see below).

The congruence rule adds a special set of conditions, which does not transfer well to graph coloring, but can be added to the SAT encoding without too much trouble. If $\tau : S \to C$ maps a term to the corresponding congruence class, these conditions can be expressed in the following way: for each two appearances $f(x_1, \ldots, x_n)$ and $f(y_1, \ldots, y_n)$ of a function symbol f in the input formula, $\tau(x_1) = \tau(y_1) \wedge \ldots \wedge \tau(x_n) = \tau(y_n)$ implies that $\tau(f(x_1, \ldots, x_n)) = \tau(f(y_1, \ldots, y_n))$, i.e. the two terms must be in the same congruence class if their arguments are pairwise congruent. This is simply a statement of the congruence rule. In the graph coloring analogy, this would be equivalent to adding a bunch of conditions which say that if certain pairs of vertices are assigned the same color, then some other pair must also be assigned the same color. For example, if the input formula is $g(a, b) = b \wedge g(c, d) \neq a$, the set of congruence classes obtained after congruence closure would be $\{\{a\}, \{b, g(a, b)\}, \{c\}, \{d\}, \{g(c, d)\}\}$. Translating this into a graph coloring problem where vertex v_1 corresponds to class $\{a\}$, v_2 to $\{b, g(a, b)\}$, etc, the congruence rule would impose the condition that if vertices v_1 and v_3 are the same color (i.e. a and c are congruent), and v_2

and v_4 also the same color, then v_2 and v_5 must be the same color too. Such conditions are of course not part of the classical graph coloring problem, but they can be just added into the SAT encoding.

In summary, the algorithm is the following:

1. Apply congruence closure, disregarding the domain cardinality constraint. If the resulting congruence classes are in conflict with the input disequalities, return `unsat`.

2. If the number of congruence classes is smaller than or equal to the required size of the domain, we already have a model, return `sat`.

3. Encode the congruence classes, the disequalities, and the domain cardinality constraint to SAT (see below).

4. If the SAT solver returns `unsat`, return `unsat`. Otherwise, a model can be extracted from the SAT model (see below), return `sat`.

Note that SAT is incremental: points 3 and 4 could be done progressively, with the graph coloring problem encoded first, then adding needed congruence constraint on demand whenever they are not satisfied.

6.6.1 SAT encoding

To encode the problem into SAT, start by encoding the graph coloring part. If there are $|\mathcal{D}|$ elements to assign to $|C|$ congruence classes, this can be done by creating $|\mathcal{D}||C|$ propositions $p_{i,j}$, with $1 \leq i \leq |\mathcal{D}|$ and $1 \leq j \leq |C|$, indicating whether element i is assigned to class j. The following conditions then need to be encoded:

- At least one element must be assigned to each class ($|C|$ clauses of length $|\mathcal{D}|$): $\bigwedge_{j=1}^{|C|} \left(\bigvee_{i=1}^{|\mathcal{D}|} p_{i,j} \right)$
- At most one element can be assigned to each class ($|C||\mathcal{D}|(|\mathcal{D}|-1)/2$ binary clauses): $\bigwedge_{j=1}^{|C|} \bigwedge_{1 \leq i < k \leq |\mathcal{D}|} (\neg p_{i,j} \vee \neg p_{k,j})$
- For each pair of classes j and k that must be different (because some element of class j is different from some other element of class k), encode this condition as $|\mathcal{D}|$ binary clauses: $\bigwedge_{i=1}^{|\mathcal{D}|} (\neg p_{i,j} \vee \neg p_{i,k})$

This encodes the graph coloring problem. One can note the similarity with the propositional encoding of Sudokus (cf. Section 3.1.2). This is because each Sudoku is essentially an instance of graph 9-coloring.

Now, the conditions resulting from the congruence rule need to be added. For each pair of appearances of a function symbol in the input formula, denoted $f(x_1, \ldots, x_n)$ and $f(y_1, \ldots, y_n)$, construct a set $args = \{\{c(x_1), c(y_1)\}, \ldots, \{c(x_n), c(y_n)\}\}$, where c maps a term to the numerical index of the corresponding congruence class. The constraint can then be encoded as:

$$\left(\bigwedge_{\{j,k\} \in args} \left(\bigvee_{i=1}^{|\mathcal{D}|} (p_{i,j} \wedge p_{i,k})\right)\right) \Rightarrow \left(\bigvee_{i=1}^{|\mathcal{D}|} p_{i,a} \wedge p_{i,b}\right)$$

where $a = c(f(x_1, \ldots, x_n))$, $b = c(f(y_1, \ldots, y_n))$, and j and k are taken in any order. If both i-th arguments are congruent, i.e. if $c(x_i) = c(y_i)$, then the corresponding pair can simply be removed from $args$.

State-of-the-art solvers expect to receive the formula directly in conjunctive normal form (CNF), which is not the case here. Converting the formula naively using distributive properties and De Morgan's law leads to a CNF of exponential size in $|\mathcal{D}|$. Instead, an equisatisfiable CNF of comparable size can be obtained by introducing some new propositions. This is akin to a Tseitin transformation.

Start by introducing $|C|(|C| - 1)/2$ propositions $q_{j,k}$, with $j < k$, such that $q_{j,k}$ is true when classes j and k are assigned the same element (i.e. when the classes should be merged). Using these, the condition becomes $\left(\bigwedge_{\{j,k\} \in args} q_{j,k}\right) \Rightarrow q_{a,b}$, which is a clause (of size at most $arity(f) + 1$), since it is equivalent to:

$$q_{a,b} \vee \bigvee_{\{j,k\} \in args} \neg q_{j,k}$$

Unlike before, when their order did not matter, j and k now must be taken in the order such that $j < k$, otherwise $q_{j,k}$ does not exist. In practice, not all $q_{j,k}$ need to be created: only the ones such that $\{j, k\} \in args$, as well as $q_{a,b}$, are needed. These new propositions also need to be linked to the previous ones:

$$q_{j,k} \Leftrightarrow \left(\bigvee_{i=1}^{|\mathcal{D}|} (p_{i,j} \wedge p_{i,k})\right)$$

This is again not a CNF. The right to left implication can be encoded as $|\mathcal{D}|$ ternary clauses (for each $q_{j,k}$):

$$\bigwedge_{i=1}^{|\mathcal{D}|} ((p_{i,j} \wedge p_{i,k}) \Rightarrow q_{j,k}) \longleftrightarrow \bigwedge_{i=1}^{|\mathcal{D}|} (\neg p_{i,j} \vee \neg p_{i,k} \vee q_{j,k})$$

The left to right implication requires a bit more work. For each $q_{j,k}$, create $|\mathcal{D}|$ new propositions $r_{i,j,k}$ (with $1 \leq i \leq |\mathcal{D}|$). The implication can then be encoded using two sets of clauses:

- $q_{j,k} \Rightarrow \left(\bigvee_{i=1}^{|\mathcal{D}|} r_{i,j,k}\right) \longleftrightarrow \neg q_{j,k} \vee \bigvee_{i=1}^{|\mathcal{D}|} r_{i,j,k}$
- $\bigwedge_{i=1}^{|\mathcal{D}|}(r_{i,j,k} \Rightarrow (p_{i,j} \vee p_{i,k})) \longleftrightarrow \bigwedge_{i=1}^{|\mathcal{D}|}(\neg r_{i,j,k} \vee p_{i,j} \vee p_{i,k})$

For each $q_{j,k}$, this corresponds to one clause of size $|\mathcal{D}| + 1$ and $|\mathcal{D}|$ ternary clauses.

This concludes the SAT encoding, in conjunctive normal form, of the satisfiability problem in the theory of equality and uninterpreted functions under domain cardinality constraints. If the SAT solver returns `unsat`, the formula is unsatisfiable. It it returns `sat`, a model can be extracted from the SAT model. To do this, simply merge the congruence classes which have been assigned the same element, i.e. merge classes j and k if there is an element i such that $p_{i,j}$ and $p_{i,k}$ are both true . If $q_{j,k}$ is defined, this variable also tells whether the classes should be merged. The resulting set of congruence classes indicates all the (dis)equalities that should hold.

6.7 Extensions for multiple sorts

The previous algorithm only works in the case where there is a single finite domain. It is possible to have problems in which multiple domains are involved. Some of the domains can subjected to cardinality constraints, other domains may be infinite. Of course, only elements within the same domain can be equal. If we were working with the theory of equality alone, those domains would thus be completely disconnected. It would then be enough to apply the previous algorithm once for each domain, and the overall formula would be satisfiable if and only if there is a satisfying assignment for each individual domain. When functions are added into the mix, interactions between domains become possible, and treating each domain completely independently is not possible anymore. For example, there could be a function $f : \mathcal{D}_1 \times \mathbb{Z} \longrightarrow \mathcal{D}_2$, with $|\mathcal{D}_1| = 7$ and $|\mathcal{D}_2| = 64$. The congruence rule would result in constraints that link elements from both finite and infinite domains.

6.7.1 Isolating the domains

A first idea might be to reuse the single sort algorithm on each individual domain, but first computing the additional equalities and disequalities stemming from interactions with other domains. For example, take the problem

$a \neq b \wedge a \neq c \wedge f(b) \neq f(c)$, where $a, b, c \in \mathcal{D}$, $|\mathcal{D}| = 2$ and $f : \mathcal{D} \longrightarrow \mathbb{Z}$. $f(b) \neq f(c)$ imposes $b \neq c$. Taking this into account, we can apply the single sort algorithm on $\mathcal{D}$ with the formula $a \neq b \wedge a \neq c \wedge b \neq c$. This is unsatisfiable for $|\mathcal{D}| = 2$, so the original problem is unsatisfiable as well.

Unfortunately, it is not always so simple to track down the constraints stemming from inter-domain interactions. Had the formula been $a \neq b \wedge a \neq c \wedge f(b, d) \neq f(c, b)$, with $f : \mathcal{D} \times \mathcal{D} \longrightarrow \mathbb{Z}$, the constraint resulting from the last disequality would have been $b \neq c \vee d \neq b$. This being a disjunction, it cannot just be added to the input for the single sort algorithm: the theory solver only works on conjunctions of (dis)equalities. A more powerful solver would be required to be able to work with that kind of constraint. Typically, a full blown SMT solver would be used to deal with such formulas. Worse, some conditions might even be impossible to isolate to a single domain. Take the problem $a \neq b \wedge a \neq c \wedge A \neq B \wedge A \neq C \wedge f(b, B) \neq f(c, C)$, where $a, b, c \in \mathcal{D}_1$, $A, B, C \in \mathcal{D}_2$, $|\mathcal{D}_1| = |\mathcal{D}_2| = 2$, and $f : \mathcal{D}_1 \times \mathcal{D}_2 \longrightarrow \mathbb{Z}$. The last disequality gives the disjunction $b \neq c \vee B \neq C$, which cannot be exclusively associated to any of the two domains. There is always the possibility of guessing which branch of the disjunction is true, but as these types of clauses add up, the number of possible guesses grows exponentially. With these issues, trying to eagerly encode all the inter-domains constraints into constraints for individual domains seems like a bad idea.

6.7.2 Lazy evaluation of inter-domain constraints

Instead of trying to guess in advance which constraints will hold, as suggested in the last section, another approach may be to start by ignoring the inter-domain constraints altogether. First try to find a model for each domain, respecting only the intra-domain constraints, i.e. only taking into account the input (dis)equalities on this specific domain, along with the axioms of reflexivity, symmetry, transitivity, and only take into account the congruence rule for functions that stay entirely within this domain ($f : \mathcal{D}^n \longrightarrow \mathcal{D}$ for some positive integer n). Then, if a model is found for this domain, see if it fits with the rest of the constraints. If not, add a clause excluding this model to the SAT encoding. Repeat until a model is found for each domain, respecting all of the inter-domain constraints. If at some point we run out of models to try, the problem is unsatisfiable. Here is a potential algorithm:

1. First execute classical congruence closure, as would be done without any cardinality constraints. The equalities derived here are all necessary. If there is a conflict with the input disequalities, return `unsat`.

2. Select a finite domain for which a model has not been found yet, i.e. a domain on which the number of congruence classes is greater than the cardinality of the domain. If there is no such domain, return `sat`.

3. Encode the intra-domain constraints in SAT.

4. Run the SAT solver to try to find a model for this finite domain.

5. If no model is found, go back to the previous finite domain for which a model was found. Negate that model in the SAT encoding and backtrack to remove the equalities propagated from that model, then try again from step 4. If there is no such previous domain, return `unsat`. Otherwise, propagate the equalities of the model using congruence closure.

6. If propagating leads to a conflict with the input disequalities, backtrack, negate the model in SAT, and try again from step 4. If not, go to step 2.

This algorithm has the advantage that the SAT solver does not need to start from scratch after each model it tries: if the SAT solver is incremental, a clause negating the failed model can simply be added to the input, and the SAT solver can continue with everything it had already learned.

The hope is that most of the conflicts occur within the domain, where they can be reasoned about and resolved by the SAT solver. If all the conflicts stem from inter-domain constraints, this is no different from a brute force approach: try every combination until a global model is found.

Unfortunately, it is quite easy to come up with a problem where this is the case. Take the quantifier-free encoding of the Sudoku problem. If the numbers are represented as part of a finite sort $\mathcal{D} = \{1, \ldots, 9\}$, then everything is fine: the problem is just a statisfiability problem in the theory of uninterpreted functions, with a single domain of size $|\mathcal{D}| = 9$. The problem can be solved using the single sort algorithm of Section 6.6, or equivalently, the above algorithm. The Sudoku will essentially just be re-encoded in SAT, where it can (hopefully) be solved efficiently. Now, tweak this encoding slightly: declare a function $f : \mathcal{D} \longrightarrow \mathbb{Z}$, and instead of encoding the row, column, and block constraints in the form $A(i, j) \neq A(k, l)$, encode them over the integers, through this new f function: $f(A(i, j)) \neq f(A(k, l))$. For example, $A(1, 1) \neq A(1, 6)$ becomes $f(A(1, 1)) \neq f(A(1, 6))$. The constraints pretty much stay the same, but this slight tweak converts them from being intra-domain to inter-domain. If we run this algorithm on this new encoding, things are very different: the row, column, and block constraints are not

intra-domain constraints, so they do not get encoded into SAT. The SAT solver comes up with a model for the constraints that were encoded: any 9 by 9 grid of numbers from 1 to 9 works, as long as it respects the numbers that were already present. Then, this model is checked against the inter-domain constraints, meaning the grid is checked against the Sudoku rules. The chances of it working are of course very low, so the model is negated, and the SAT solver is asked to come up with another one. It tries another random grid, and the process repeats until the right grid is found. Needless to say, this is extremely inefficient, and unlikely to solve the Sudoku in any reasonable amount of time.

There are some ways to improve this algorithm which can mitigate this kind of problem. The first one is that all inter-domain constraints do not need to be evaluated lazily. Cases such as $f(a) \neq f(b)$, with $f : \mathcal{D}_1 \longrightarrow \mathcal{D}_2$ can be converted to constraints on $\mathcal{D}_1$: $a \neq b$. More generally, each time inter-domain constraints lead to intra-domain consequences, those intra-domain consequences can be encoded in SAT. This is a way to reduce the number of possible models for the SAT solver. This improvement alone would solve the problem of the above encoding of Sudoku: all the constraints could be converted back to constraints on $\mathcal{D}$, and the SAT solver would again only have one model to try: the correct grid. There is however a trade-off to be made here: computing *all* of the consequences of inter-domain constraints can be rather impractical. On top of that, some of them might overlap. Some more work needs to be done in order to figure out which constraints are worth spending time on to encode in advance, and which ones should be left to be evaluated lazily.

A second possible improvement would be be to negate multiple models at once. When blindly propagating the equalities from the SAT model, finding a conflict tells us very little: the only thing that is learned is that the model was wrong, and another model which differs even slightly might be right. Instead, we could try to track down *why* the model does not work. Imagine a model containing the equalities $a = b$ and $c = d$, and one of the inter-domain constraints being $f(a, c) \neq f(b, d)$. That model would obviously lead to a conflict, but we know this conflict is due to the conjunction $a = b \wedge c = d$. Instead of a clause negating the whole model, the clause $a \neq b \vee c \neq d$ could be added to SAT. This would not only get rid of this specific model, but also all the models where $a = b \wedge c = d$, which would all fail for the same reason. In general, the smaller an explanation is for why the model fails (in terms of the number of equalities that lead to the conflict; in the previous example, that would be two), the more models can be excluded. Adding some reasoning capable of tracking down why models fail, i.e. which are the equalities that lead to a conflict, could greatly improve the performances

of this algorithm. Fortunately, there already exist versions of congruence closure capable of efficiently producing such small explanations [40].

6.7.3 Problems with isolating the domains

The last algorithm might not appear very convincing. Even with the given improvements, there is still potential for inter-domain constraints to be treated with what is essentially a brute-force approach, where reasoning about them may instead be possible and a lot more effective. The system as a whole also appears quite complex, resembling an SMT procedure with its interaction with the SAT solver and lazy evaluation of the constraints that are not encoded. Overall, trying to reason about the different domains independently may be a poor idea when they can be interconnected in such complex ways.

Actually, it can be shown that imposing cardinality constraints on one domain can influence the satisfiability of cardinality constraints on another domain. This means both domains can never be reasoned about fully independently, as the constraints are satisfiable for each domain on their own, but not when the domains are taken together. This phenomenon appears in an example from earlier: $a \neq b \wedge a \neq c \wedge A \neq B \wedge A \neq C \wedge f(b, B) \neq f(c, C)$, with $a, b, c \in \mathcal{D}_1$, $A, B, C \in \mathcal{D}_2$, and $f : \mathcal{D}_1 \times \mathcal{D}_2 \longrightarrow \mathbb{Z}$. Each domain contains at least two elements: a and b, or A and B. A third element for c (or C) is not necessary if c can be equal to b (or if C can be equal to B). However, the last inequality imposes $b \neq c \vee B \neq C$. If we impose $|\mathcal{D}_1| \leq 2$, the problem is satisfiable by having $b = c$. The last inequality can be respected by taking $B \neq C$. Similarly, if we impose $|\mathcal{D}_2| \leq 2$, the problem is satisfiable by taking $B = C$ and $b \neq c$. However, if we try to take both constraints simultaneously, i.e. $|\mathcal{D}_1| \leq 2$ *and* $|\mathcal{D}_2| \leq 2$, the problem is not satisfiable anymore, as we need $b = c \wedge B = C$, but this violates the last inequality. This means that imposing the constraint on one domain prevents the constraint on the other domain from being satisfiable.

6.7.4 Full SAT encoding of the multi-sorted problem

So far, the approach to the problem with multiple domains was to split it into a problem on each individual domain, and try to reconcile the solutions to get a solution to the global problem. As the last section suggests, this approach can be problematic. Reasoning on all the domains taken together has the potential to be really helpful, as it could avoid trying every solution for every domain until common ground is found (or not). Information learned on one domain could be propagated to another domain and reduce the search space drastically. In the last example, with cardinality constraint $|\mathcal{D}_1| \leq 2$,

it can be learned that $b = c$, and thus $B \neq C$. Important information about elements in $\mathcal{D}_2$ is thus learned, coming from constraints on $\mathcal{D}_1$. Using this information, none of the models where $B = C$ have to be tried.

One way to have this inter-domain reasoning could simply be to encode the whole problem in SAT, making the SAT solver work on every domain and their interactions at once. Since the problem with multiple sorts is also in NP, this must be possible with a polynomial encoding. As it turns out, extending the encoding of the single sort problem to work with multiple sorts is fairly straightforward.

One of the challenges is that some domains may have infinite cardinality, which could not be encoded in the way finite domains are for the problem with a single sort, since it would lead to infinitely many propositions $p_{i,j}$. Fortunately, there is no need to keep track of all the elements of the domain: only the terms given as input are interesting. Since there is no limit on the number of elements, there is no need to check that an element is available for each of the terms (a term can always be different from every other term, as long as there are finitely many terms). We can simply keep track of which terms are equal. This is already partly done in the single sort encoding with the use of the $q_{j,k}$ propositions.

Getting rid of the $p_{i,j}$ propositions lifts the domain cardinality constraints, but another important rule is lost: transitivity. Transitivity was a natural consequence of encoding which element is assigned to a term (or to congruence class): if two classes j and k are equal, then they are both assigned the same element i ($p_{i,j}$ and $p_{i,k}$ are both true). If a third class l is also equal to class k, it is also assigned element i ($p_{i,l}$ is also true). Since both classes j and l are assigned the same element (i), they are automatically assumed to be equal, and transitivity is respected. This is not the case anymore if we only track which terms/classes are equal: we could have class j equal to class k and class k equal to class l ($q_{j,k}$ and $q_{k,l}$ both true), but class j different from class l ($q_{j,l}$ false). This can be remedied by imposing transitivity explicitly, with clauses of the form:

$$q_{j,k} \wedge q_{k,l} \Rightarrow q_{j,l} \longleftrightarrow \neg q_{j,k} \vee \neg q_{k,l} \vee q_{j,l}$$

In domain with $|C|$ congruence classes, there would be $\binom{|C|}{2}(|C| - 2) = |C|(|C| - 1)(|C| - 2)/2$ such ternary clauses. Reflexivity is implied in both encodings ($q_{j,j}$ is not even defined). Symmetry as well: if $j < k$, $q_{j,k}$ represents both class j equal to class k and class k equal to class j; $q_{k,j}$ is not defined. The congruence rule needs to be encoded explicitly, as is done in the single sort encoding.

6.7.5 Features of the theory solver

This section describes the features that a theory solver using the previous algorithm would have, and the ones it would lack. A theory solver can have a few desirable features [15], namely:

- Model generation: when the set of literals is satisfiable, the theory solver can provide a model.
- Conflict set generation: when the set of literals is unsatisfiable, the theory solver can provide a (possibly minimal) subset of literals which cause the conflict.
- Incrementality: literals can be added without the solver needing to start from scratch.
- Backtrackability: the theory solver can efficiently undo steps and return to a previous state.
- Deduction of unassigned literals: the theory solver can deduce the value of some unassigned literal if it is a consequence of the given set of literals.
- Deduction of interface equalities: the theory solver can deduce (disjunctions of) equalities which are consequences of the given set of literals.

Modern congruence closure algorithms have most of these features [38]: they can efficiently produce models, produce small conflict sets, deduce all entailed equalities, and they are incremental and backtrackable. The algorithm described in the last section keeps all of these properties when the domain cardinality constraints are not too tight, i.e., when the cardinality of each sort is larger than the number of congruence classes after applying congruence closure. In those cases, SAT does not need to be called, and the algorithm correspond to pure congruence closure.

When at least one of the sorts has more congruence classes than allowed by its cardinality constraint, the problem is encoded in SAT. Model generation is intact, which is good. SAT solvers being backtrackable and incremental, those features are kept as well: when a literal is added, the corresponding propositions and clauses can be added incrementally to the SAT encoding. SAT however does not perform any deduction: only a model is generated, and there is no guarantee that any of the equalities in the model have to hold.

Equalities may still be deduced in the following way: when a model is found, take an equality from the model (which was not already deduced by

congruence closure), and add its negation to the set of literals. If the problem becomes unsatisfiable, this equality was a consequence of the original set of literals (and the cardinality constraints). Since each equality has to be tested individually, this is likely to be inefficient in practice. When cardinality constraints are involved, the theory also becomes non-convex, meaning disjunctions of equalities could be deduced as well. The same process may be used, but as the number of disjunctions of equalities is exponential in the number of equalities, this is definitely going to be inefficient.

When a new input equality is added incrementally, congruence closure may be used in parallel with SAT to deduce the equalities which do not rely on cardinality constraints. If enough classes are merged within congruence closure when doing this, one may consider starting the SAT encoding from scratch. This does not preserve incrementality in the SAT solver, but a smaller encoding (due to less classes) may lead to less work overall.

Since deducing all entailed (disjunctions of) equalities is hard, appropriate theory combination methods for non-convex theories should be used. For example, delayed theory combination [23], or model-based theory combination [24].

There is also no obvious way to have efficient conflict set generation when SAT is involved. This along with the difficulty to deduce equalities are two important limitations to the SAT-based approach. Future research may try to find ways to mitigate this.

Chapter 7

Conclusions

This work started by exploring the capacity of current state-of-the-art SMT solvers to solve combinatorial finite domain problems such as Sudokus. It was found that, while most solvers can efficiently solve Sudokus in their quantifier-free encoding, introducing quantifiers leaves a big hit on performance. This lead to investigating approaches to automatically remove quantifiers in the presence of such finite domains.

A definition of an "effective finite domain" was established for Skolem formulas quantified over integers, allowing the elaboration of an algorithm for detecting them, and eliminating quantifiers through exhaustive instantiation on the effective finite domain. The algorithm makes use of an oracle for determining the satisfiability of *ground* formulas. Since no quantifier reasoning is necessary, this technique can be used even with solvers which do not support quantifiers, allowing them to solve the quantified problem if all quantifiers can be eliminated. On top of this, the technique is also reasonably efficient. For large problems, this allows a significant performance gain by first eliminating quantifiers and working on the dequantified encoding, compared to directly working with the quantified encoding.

Future work on this technique includes testing it on various benchmarks for finite domain problems (such as those provided by Clearsy for the BLaSST project [1]), evaluating when this procedure is worth using and when it should be avoided, implementing it efficiently in a state-of-the-art solver, and investigating techniques for partial instantiation when the finite domain is too large for exhaustive instantiation to be practical. It may also be worthwhile to investigate whether similar techniques can be applied to sorts other than the integers.

In the second part of this work, algorithms were developed for working

[1] `https://www.clearsy.com/en/research-and-development/blasst/`

on the theory of uninterpreted functions with domain cardinality constraints. The satisfiability problem for a set of literals in this theory was shown to be NP-complete. SAT-based algorithms extending classical congruence closure were then developed and analyzed theoretically. For domains that are large enough, congruence closure is preserved as is, maintaining its efficiency along with efficient conflict set generation, incrementality, backtrackability, and deduction of entailed equalities. When the domain becomes too small, SAT takes over ensuring a complete reasoning. The efficiency of modern SAT solvers hopefully leads to a fast procedure for generating a model or proving unsatisfiability, while maintaining incrementality and backtrackability.

Those algorithms are left to be implemented and tested within an SMT framework. When the domain is small and SAT is used, (efficient) conflict set generation and deduction of equality are lost. The theory being non-convex, they are not easy to recover. This can make theory combination significantly harder. Future work may aim at bringing them back, at least in some limited form.

Appendix A

Source code

The code used in Chapters 4 and 5 is available on the ULiège GitLab at `https://gitlab.uliege.be/Louis.Dasnois/satisfiability-modulo-theories-for-finite-domains`. It contains various encodings of Sudokus in the SMT-LIB format [3], described in Chapter 4, and a prototype implementation, using PySMT [37], of the dequantification procedure described in Chapter 5.

Bibliography

[1] C. Barrett, L. De Moura, and A. Stump, "SMT-COMP: Satisfiability modulo theories competition," in *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings 17.* Springer, 2005, pp. 20–23.

[2] C. Barrett, L. de Moura, S. Ranise, A. Stump, and C. Tinelli, "The SMT-LIB Initiative and the Rise of SMT: (HVC 2010 Award Talk)," in *Hardware and Software: Verification and Testing: 6th International Haifa Verification Conference, HVC 2010, Haifa, Israel, October 4-7, 2010. Revised Selected Papers 6.* Springer, 2011, pp. 3–3.

[3] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017.

[4] D. Beyer, M. Dangl, and P. Wendler, "A unifying view on SMT-based software verification," *Journal of automated reasoning*, vol. 60, no. 3, pp. 299–335, 2018.

[5] R. Mukherjee, D. Kroening, and T. Melham, "Hardware verification using software analyzers," in *2015 IEEE Computer Society Annual Symposium on VLSI.* IEEE, 2015, pp. 7–12.

[6] K. R. M. Leino, "Automating theorem proving with SMT," in *International Conference on Interactive Theorem Proving.* Springer, 2013, pp. 2–16.

[7] L. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-based bounded model checking for embedded ANSI-C software," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 957–974, 2011.

[8] M. Bofill, J. Suy, and M. Villaret, "A system for solving constraint satisfaction problems with SMT," in *Theory and Applications of Satisfiability Testing–SAT 2010: 13th International Conference, SAT 2010,*

Edinburgh, UK, July 11-14, 2010. Proceedings 13. Springer, 2010, pp. 300–305.

[9] J. Vanegue, S. Heelan, and R. Rolles, "SMT solvers in software security." *WOOT*, vol. 12, pp. 9–22, 2012.

[10] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 2008, pp. 367–381.

[11] S. Szymoniak, O. Siedlecka-Lamch, A. M. Zbrzezny, A. Zbrzezny, and M. Kurkowski, "Sat and smt-based verification of security protocols including time aspects," *Sensors*, vol. 21, no. 9, p. 3055, 2021.

[12] H. K. Sahu, N. R. Pillai, I. Gupta, and R. K. Sharma, "SMT solver-based cryptanalysis of block ciphers," *SN Computer Science*, vol. 1, pp. 1–12, 2020.

[13] P. Fontaine, "INFO9015: Logic for computer science," University of Liège, 2022.

[14] P. Fontaine and B. Boigelot, "INFO0060: Introduction to computer systems verification," University of Liège, 2022.

[15] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, 2009, ch. 26, pp. 825–885.

[16] P. Fontaine, "Techniques for verification of concurrent systems with invariants," Ph.D. dissertation, Institut Montefiore, Université de Liège, Belgium, Sep. 2004.

[17] G. S. Tseitin, "On the complexity of derivation in propositional calculus," *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pp. 466–483, 1983.

[18] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971, pp. 151—-158.

[19] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of satisfiability*. IOS press, 2021, pp. 133–182.

[20] DIMACS, "Satisfiability suggested format," 1993. [Online]. Available: http://archive.dimacs.rutgers.edu/pub/challenge/satisfiability/doc/

[21] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL(T): Fast decision procedures," in *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings 16*. Springer, 2004, pp. 175–188.

[22] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 2, pp. 245–257, 1979.

[23] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani, "Efficient theory combination via boolean search," *Information and Computation*, vol. 204, no. 10, pp. 1493–1525, 2006.

[24] L. de Moura and N. Bjørner, "Model-based theory combination," *Electronic Notes in Theoretical Computer Science*, vol. 198, no. 2, pp. 37–49, 2008.

[25] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A theorem prover for program checking," Hewlett Packard Laboratories, Tech. Rep. HPL-2003-148, Jul. 23 2003.

[26] L. M. de Moura and N. Bjørner, "Efficient E-matching for SMT solvers," in *CADE*, ser. LNCS, vol. 4603, 2007, pp. 183–198.

[27] Y. Ge and L. M. de Moura, "Complete instantiation for quantified formulas in satisfiabiliby modulo theories," in *CAV*, ser. LNCS, vol. 5643, 2009, pp. 306–320.

[28] A. Reynolds, C. Tinelli, A. Goel, S. Krstic, M. Deters, and C. W. Barrett, "Quantifier instantiation techniques for finite model finding in SMT," in *CADE*, ser. Lecture Notes in Computer Science, vol. 7898. Springer, 2013, pp. 377–391.

[29] A. Reynolds, C. Tinelli, and L. M. de Moura, "Finding conflicting instances of quantified formulas in SMT," in *FMCAD*, 2014, pp. 195–202.

[30] H. Barbosa, P. Fontaine, and A. Reynolds, "Congruence closure with free variables," in *TACAS*, ser. LNCS, vol. 10206, 2017, pp. 214–230. [Online]. Available: Barbosa1b.pdf

[31] A. Reynolds, H. Barbosa, and P. Fontaine, "Revisiting enumerative instantiation," in *TACAS*, ser. LNCS, vol. 10806, 2018, pp. 112–131.

[32] M. Janota, H. Barbosa, P. Fontaine, and A. Reynolds, "Fair and adventurous enumeration of quantifier instantiations," in *FMCAD*. IEEE, 2021, pp. 256–260. [Online]. Available: https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_35

[33] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[34] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli *et al.*, "cvc5: A versatile and industrial-strength SMT solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 415–442.

[35] B. Dutertre, "Yices 2.2," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 737–744.

[36] "cvc5 smt-comp scripts." [Online]. Available: https://github.com/cvc5/cvc5/tree/main/contrib/competitions/smt-comp

[37] M. Gario and A. Micheli, "PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms," in *SMT workshop*, vol. 2015, 2015.

[38] R. Nieuwenhuis and A. Oliveras, "Fast congruence closure and extensions," *Information and Computation*, vol. 205, no. 4, pp. 557–580, 2007.

[39] P. Bahr, "Implementation of a fast congruence closure algorithm," Technische Universität Dresden, Tech. Rep., 2007.

[40] R. Nieuwenhuis and A. Oliveras, "Proof-producing congruence closure," in *International Conference on Rewriting Techniques and Applications*. Springer, 2005, pp. 453–468.