PARTIAL GROUNDING

by

Brendan Guild

BSc, Simon Fraser University, 2005

a Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
Spring 2011

APPROVAL

Name: Brendan Guild

Degree: Master of Science

Title of Thesis: Partial Grounding

Examining Committee: Dr. Oliver Schulte
Associate Professor, Computing Science
Chair

Dr. Evgenia Ternovska
Associate Professor, Computing Science
Senior Supervisor

Dr. David G. Mitchell
Associate Professor, Computing Science
Supervisor

Dr. James P. Delgrande
Professor, Computing Science
SFU Examiner

Date Approved: 26 April 2011

Declaration of
Partial Copyright Licence

Abstract

Model expansion is the task underlying many paradigms for declarative solving of computationally hard search problems such as scheduling, planning, and formal verification. We develop a technique to solve model expansion tasks involving arithmetic and other infinite secondary structures. Unlike previously developed methods, our technique produces quantifier-free formulas suitable for tools that are specially designed for arithmetic such as satisfiability modulo theories (SMT) solvers. The novelty is in that our method leaves part of the specification not grounded. We design new algorithms which first perform partial grounding, and then take a quantifier-free formula and produce a formula suitable for solving using SMT. An SMT solver is called on the resulting formula. We describe how we constructed a software tool for partial grounding and list several example inputs and the results of using the Yices SMT solver on the outputs.

Acknowledgments

I thank my supervisor, Eugenia Ternovska, who is responsible for the existence of this paper and any amount of professionalism that it contains.

I also thank Dr. David G. Mitchell for important technical advice.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Search problems are a broad class of problems that includes instances in graph theory, planning, and optimization. There are numerous tools that can be used for solving such problems such as SAT solvers [102, 13] for propositional logic, Satisability Modulo Theories [26, 8, 103] for specialized fragments of first-order logic, and Answer Set Programming [53] solvers for logic programming.

Model expansion (MX) is a very intuitive way to formally represent a search problem. In this way one can express a wide variety of problems using, for example, first-order logic with extensions. It is useful to specialize automated solvers for specific varieties of model expansion to improve performance, and then we can have both an intuitive problem representation and an efficient solver. For example, arithmetic is commonly involved in practical problems and computers are made to do arithmetic, so it is natural to build an automated solver so that it uses a special strategy when arithmetic is present, especially since structures that represent arithmetic tend to be infinite.

Satisfiability Modulo Theories (SMT) is the problem of using a first-order theory to decide satisfiability of a first-order formula. Each SMT solver is constructed for a particular set of theories and fragments of first-order logic that can be solved efficiently. It deliberately separates a theory from the input formula so that solving algorithms can be specialized for each theory. This should make them ideal for solving problems that involve arithmetic, as well as potentially any problem involving one of the supported theories, but using an SMT solver directly is far more technically complex than expressing a search problem as model expansion.

A SAT solver is another problem solving tool, more commonly used than SMT solvers.

The SAT solver finds a satisfying assignment for propositional variables in a propositional logic formula. This task is relatively simple compared to the task of a SMT solver. Since SAT solvers need not deal with functions, predicates, or theories, the designer of a SAT solver can focus a large amount of effort into shaving time and memory off the solving procedure. Even so, for a SAT solver to solve a problem involving arithmetic, it must run an algorithm that treats arithmetic just like any other problem. Without predicates or functions, you cannot represent arithmetic in propositional logic in a way that makes it directly recognizable as arithmetic.

When you have a problem involving arithmetic, if you reduced it to a SAT problem, you can expect your SAT solver to solve it in highly efficient but naive way. We will design a way to reduce model expansion problems represented in first-order logic with arithmetic to SMT problems so that we can take advantage of solvers that a tailored to exactly the sort of problem we are trying to solve.

For our purposes, grounding is taking a first-order formula and an instance structure and constructing a ground formula whose models are the model expansions of the original formula and structure. This is the usual method for solving model expansion problems as it allows us to construct a propositional formula for a SAT solver. For example, you may ground by substituting every possible constant for each variable, and then substitute a propositional variable for each distinct ground atom. Instead of constructing a ground formula, we will construct a formula without quantifiers but still potentially having variables, as if stopping the grounding process partway through. We call this *partial grounding* and it is very useful for reducing model expansion to SMT.

Depending on the theory being used, SMT solvers almost always accept a fragment of first-order logic that does not include quantifiers, so partial grounding is the natural first step. The fragments also tend to have very restricted vocabularies, often allowing only the vocabulary of the theory itself. This leads us to the next step, where we perform substitutions to eliminate all of the vocabulary that we are using for model expansion and are left with only the vocabulary that our SMT solver will accept, such as arithmetic vocabulary.

This thesis goes into depth on ways to construct a tool for partial grounding of a model expansion problem with a given instance, how to use a partial grounding to construct an input for an SMT solver, and how to interpret the output of an SMT solver to find a solution to the original model expansion problem. Our goal is to allow practical search problems to

be expressed compactly and easily as a model expansion problem, while at the same time solving the problem efficiently.

1.1 Contributions

The following are the contributions that we have made in the area of computational logic for the purpose of solving elaborate model expansion problems more easily, especially model expansion involving arithmetic.

1. Definition of Partial Grounding

   We define partial grounding in terms of a formula ϕ and a structure $\mathcal{A}$. A partial grounding is a quantifier free formula that represents the same model expansion problem with $\mathcal{A}$ as the original formula ϕ represented with $\mathcal{A}$. When the free variables are considered to be existentially quantified, any expansion of $\mathcal{A}$ that satisfies ϕ also satisfies the partial grounding.

2. Algorithm for Finding a Partial Grounding

   We describe two algorithms for constructing a partial grounding of a given formula and structure. One algorithm is a simple direct approach to illustrate the concept. The second algorithm makes use of relational algebra in an attempt to produce a smaller partial grounding.

3. Technique for Solving Model Expansion using Satisfiability Modulo Theories

   We show the usefulness of partial grounding by using it to solve model expansion problems efficiently with a satisfiability modulo theories solver. Given a partial grounding of a model expansion problem, the algorithm constructs a satisfiability modulo theories formula whose solution provides one possible solution for the model expansion problem. By using satisfiability modulo theories, we can take advantage of the ability of existing solvers to solve problems involving arithmetic and all of the other various theories for which SMT solvers have been constructed. We also describe how one might add to the algorithm to allow expansion functions in the partial grounding.

4. Tool for Constructing Satisfiability Modulo Theories Problems

We have constructed various versions of a computer program that takes a model expansion problem formula and structure, constructs a partial grounding, and then constructs a satisfiability modulo theories formula as output. The versions use various software designs and have various features in both output formulas and the languages that they accept. The first version is based upon a previously existing grounding software for use with a SAT solver and we have heavily modified it to cause it to perform partial grounding for use with an SMT solver. The second version was made entirely for partial grounding for SMT, with the intention of avoiding some of the unfortunate software design elements from the first grounder, as well as adding the capability of accepting expansion functions in the input and simplifying the output SMT formula.

1.2 Outline

In Chapter 2, we describe model expansion, grounding, and the standards by which satisfiability modulo theory solvers operate. The relational algebra of extended relations from [98] is explained. Extended relations are relations that contain an extra column that holds a formula for each row, and all of the relational algebra operations are extended to operate on this column. The algorithm from [98] for grounding a first order formula using relational algebra is provided for comparison with the similar partial grounding algorithm in Chapter 3.

In Chapter 3, partial grounding is defined and explained. An algorithm for finding a partial grounding of a formula is introduced and its correctness is proven. A simple algorithm is given, and we also show an elaborate algorithm which takes advantage of relational algebra to produce a smaller output.

In Chapter 4, an algorithm for constructing an SMT formula for solving a model expansion problem using partial grounding is introduced and its correctness is proven. We explain how to find a solution to the model expansion problem using the solution provided by an SMT solver when given a constructed SMT formula.

In Chapter 5, we explain the design and details of construction of a software tool for doing partial grounding and constructing the SMT formula as described in the previous chapter. We also explain some of the changes that were made for the second version of the partial grounding tool.

In Chapter 6, we provide examples of model expansion problems and the resulting partial groundings and SMT formulas. We examine each formula and show how it is partially grounded and then converted to an satisfiability modulo theories problem.

In Chapter 7, we list the full input and the results of three applications of the grounder described in Chapter 5. The output of the grounder is too large to include, but each output is given to Yices[26], a satisfiability modulo theories solver, and the output of Yices is given in full. We explain how to interpret each output of Yices to get a solution to the original model expansion problem. For the magic square problem, we use the second version of the grounder and an axiomatization involving expansion functions, then compare the solving time of the SMT solver with the solving times of two other techniques for finding magic squares.

Chapter 2

Background

In this chapter we will explain various topics that are important for the understanding of partial grounding. In Section 2.2 we describe the way we will specify search problems. In Section 2.3 and Section 2.4 we describe an existing technique for constructing ground formulas. In Section 2.5 we describe Satisfiability Modulo Theories, a sort of problem that can be constructed from a partially grounded model expansion problem.

Throughout the paper, $:=$ is used for "denotes", $\supset$ for material implication, and $\exists\bar{x}$ for $\exists x_1 \ldots \exists x_n$, similarly for $\forall\bar{x}$.

We will use the term *literal* to mean either a first-order atom α that is not negated, or a negation followed by a first-order atom, $\neg\alpha$. A positive literal is an atom without a negation and a negative literal is an atom and its negation.

We will use a bar over an uppercase letter to indicate a tuple of relations and functions, as in the structure $(U; \bar{R})$, where U is the universe and $\bar{R}$ is the interpretations of the vocabulary.

2.1 Model Expansion

In this section we describe first-order model expansion (FO-MX), a representation for search problems. We will explain the terminology we will use with representing model expansion problems.

A vocabulary is a set of symbols. Each symbol has an arity and is either a relation symbol or a function symbol. A constant symbol is a function symbol with zero arity. For a vocabulary $\tau = \{R_1, \ldots, R_n, f_1, \ldots, f_m\}$, a τ-structure $\mathcal{A}$ is a tuple $(U; R_1^{\mathcal{A}}, \ldots, R_n^{\mathcal{A}}, f_1^{\mathcal{A}}, \ldots, f_m^{\mathcal{A}})$,

where U is the domain of $\mathcal{A}$, $R_i^{\mathcal{A}}$ is a set of k-tuples where k is the arity of relation symbol R_i, and $f_i^{\mathcal{A}}$ is a function from j-tuples to U, where j is the arity of function symbol f_i.

Definition 2.1.1 (expansion)**.** Let $\mathcal{A}$ be a σ-structure with domain A and $\mathcal{B}$ be a $(\sigma \cup \epsilon)$-structure with domain B. The structure $\mathcal{B}$ is an *expansion* of $\mathcal{A}$ if $A = B$ and for every symbol s in σ, $s^{\mathcal{A}} = s^{\mathcal{B}}$.

For a formula ϕ, we write $vocab(\phi)$ for the collection of exactly those function and relation symbols which occur in ϕ.

A structure $\mathcal{A}$ is a model of a formula ϕ if $\mathcal{A}$ satisfies ϕ by providing interpretations for $vocab(\phi)$ such that ϕ is true. Structure $\mathcal{A}$ satisfies ϕ is written as $\mathcal{A} \models \phi$.

Definition 2.1.2 (model expansion)**.** Let $\mathcal{A}$ be a σ-structure and ϕ be a formula such that $\sigma \subset \phi$. A structure $\mathcal{B}$ is a *model expansion* for ϕ over $\mathcal{A}$ if $\mathcal{B} \models \phi$ and $\mathcal{B}$ is an expansion of $\mathcal{A}$.

The vocabulary σ is called the instance vocabulary. The vocabulary of $vocab(\phi) \setminus \sigma$ is called the expansion vocabulary.

2.2 Model Expansion with Arithmetic

In order to take advantage of arithmetic in SMT problems, infinite model expansion must be used since arithmetic predicates and functions are over infinite domains. Using infinite structures naively would make grounding impossible, since formulas cannot be infinite. Therefore we will use infinite arithmetic terms in finite problems by means of *embedded model expansion*. This section describes embedded model expansion as it was described in [110].

2.2.1 Embedded Model Expansion

Embedded finite model theory (see [72, 73]), the study of finite structures whose domain is drawn from some infinite structure, was introduced to study databases that contain numbers and numerical constraints. Rather than thinking of a database as a finite structure, it is taken to be a set of finite relations over an infinite domain.

Definition 2.2.1. A structure $\mathcal{A}$ is *embedded* in an infinite *background* (or *secondary*) structure $\mathcal{M} = (U; \bar{M})$ if it is a structure $\mathcal{A} = (U; \bar{R})$ with a finite set $\bar{R}$ of finite relations

and functions, where $\bar{M} \cap \bar{R} = \emptyset$. The set of elements of U that occur in some relation or function of $\mathcal{A}$ is the *active domain of* $\mathcal{A}$, denoted $adom_{\mathcal{A}}$.

Notice that by the definition, an embedded structure is a structure. Recall that structures must have at least one element in their domains, so $adom_{\mathcal{A}}$ is never empty.

Example 2.2.1. For a simple example, consider a company database with a table containing employee numbers, salaries and pension plans. While employee numbers are abstract (summing them up makes no sense, for example), salaries and pension plans are not (taking their sum makes sense). This database is a finite structure embedded in the infinite background structure of the natural numbers with the standard arithmetic operations. Queries over embedded databases may use the database relations and the arithmetical operations whose interpretation is provided by the infinite background structure. For example, the following query (a FO formula with free variable x) returns people whose total of salary and pension plan contribution is more than \$100,000.

$$\exists s \exists p \ (employee(x, s, p) \wedge s + p \geq \$100,000).$$

In database research, embedded structures are used with logics for expressing queries, providing the interpretation of arithmetic operations in these queries. Here, they are used similarly, with logics for MX specifications (which are second order queries).

Throughout, the following conventions are used regarding the vocabulary of formulas in these logics and the associated structures.

1. σ denotes the vocabulary of the embedded structure $\mathcal{A} = (\mathcal{U}; \bar{\mathcal{R}})$, which is the instance structure.

2. ν denotes the vocabulary of an infinite background structure $\mathcal{M} = (U; \bar{M})$.

3. ε is an expansion vocabulary.

4. $\bar{R}$ and $\bar{M}$ always denote the interpretations of σ and ν, respectively. We treat $\bar{R}$ and $\bar{M}$ as tuples or as sets, depending on the context.

A formula ϕ over $\sigma \cup \nu \cup \varepsilon$ constitutes an MX specification. The model expansion task is still to expand a (now embedded) σ-structure to satisfy ϕ.

2.2.2 Double-guarded logic

A less expressive logic uses an adaptation of the guarded fragment GF_k of FO [39]. In formulas of GF_k, a conjunction of up to k atoms acts as a *guard* for each quantified variable.

Definition 2.2.2. The *k-guarded fragment* $\mathrm{GF}_k(\varepsilon)$ of FO for a given vocabulary ε is the smallest set of formulas that:

1. contains all atomic formulas not in ε;
2. is closed under Boolean operations;
3. contains $\exists\bar{x}(G_1 \wedge \ldots \wedge G_m \wedge \phi)$, provided the G_i are atomic formulas, $m \leq k$, $\phi \in GF_k$, and each free variable of ϕ appears in some G_i.

   For a formula $\psi := \exists\bar{x}(G_1 \wedge \ldots \wedge G_m \wedge \phi)$, conjunction $G_1 \wedge \ldots \wedge G_m$ is called the *guard* of the tuple of quantifiers $\exists\bar{x}$.

Since GF_k is closed under negation, universal quantification can be treated as an abbreviation in the usual way, so universal quantifiers are guarded as in $\forall\bar{x}(G_1 \wedge \ldots \wedge G_m \supset \phi)$.

In the particular case of partial grounding, universal quantifiers will not be considered an abbreviation for existential quantifiers with negation, because the goal of partial grounding is to treat universal quantifiers very differently from existential quantifiers. To create a clear distinction between universal quantifiers and existential quantifiers, negations will be required to only occur in literals and therefore never outside of a quantifier. With the quantifiers so distinguished, the guards of universal quantifiers will be used to ground the quantified variables.

Example 2.2.2. Let ε be $\{E_1, E_2\}$. The following formula is not guarded.

$$\forall x \forall y(E_1(x, y) \supset E_2(x, y)).$$

It is guarded when E_1 is replaced by P which is not in ε.

The following formula is the standard encoding of the temporal formula $Until(P_1, P_2)$.

$$\exists v_2 \ (R(v_1, v_2) \wedge P_2(v_2) \wedge \forall v_3 \ (R(v_1, v_3) \wedge R(v_3, v_2) \supset P_1(v_3)))$$

The formula is 2-guarded, that is it belongs to GF_2, but it is not 1-guarded.

The guards of GF_k can be used to limit the range of a quantified variable, which is limited to the elements in the interpretation of guard atoms. To limit the range of domain elements that may occur in expansion predicates, we may use "upper guard" axioms, which restrict the elements in expansion relations to those occurring in the interpretation of guard atoms. To formalize this, we introduce the following restriction of FO, denoted GGF_k.

Definition 2.2.3. The double-guarded fragment $GGF_k(\varepsilon)$ of FO, for a given vocabulary ε, is the set of formulas of the form $\phi \wedge \psi$, with $\varepsilon \subset vocab(\phi \wedge \psi)$, where ϕ is a formula of GF_k, and ψ is a conjunction of guard axioms, one for each symbol E of ε occurring in ψ, of the form

$$\forall \bar{x}(E(\bar{x}) \supset G_1(\bar{x}_1) \wedge \cdots \wedge G_m(\bar{x}_m)),$$

where $m \leq k$, and the set of free variables in $G_1(\bar{x}_1) \wedge \cdots \wedge G_m(\bar{x}_m)$ is precisely $\bar{x}$.

We call the guards of GF_k, which restrict the range of quantified variables, *lower guards*, and the guard axioms of $\mathrm{GGF}_k(\varepsilon)$ *upper guards*.

For expansion functions, the guard axiom is on the graph of the function. The graph of k-ary function f is the $(k+1)$-ary relation $G_f = \{(\bar{a}, b) \ : \ f(\bar{a}) = b\}$. An upper guard for f thus is of the form $\forall \bar{x} \forall y \ (f(\bar{x}) = y \supset \phi(\bar{x}, y))$, where ϕ is a conjunction of atoms.

Initially, we require all atoms providing upper and lower guards to be from the instance vocabulary, so ranges of variables and expansion predicates are explicitly limited to $adom_{\mathcal{A}}$. We later relax this restriction, adding a mechanism for "user-defined" guard relations that may contain elements not in $adom_{\mathcal{A}}$. We assume that the instance vocabulary always contains the predicate symbol $adom$, which always denotes the active domain. Then $adom(x)$ can be used as a guard atom (upper or lower)[1].

Upper and lower guards provide a logical formalization of some aspects of the type systems of some existing constraint modeling languages [91]. Lower guards correspond to declaring the types of variables, and upper guards to declaring the types of arguments to expansion predicates.

When using partial grounding to construct SMT formulas, the upper guards will not be needed, and will be treated as the same as any other part of the formula.

[1]The relation which corresponds to the active domain is definable with respect to each instance structure, but the defining FO formula requires disjunctions, thus cannot be used as a guard and the predicate symbol $adom(x)$ is necessary.

For writing MX specifications for embedded structures, we extend the logic $\mathrm{GGF}_k(\varepsilon)$ with vocabulary for a fixed background structure $\mathcal{M}$. We will talk about "$\mathrm{GGF}_k(\varepsilon)$ MX specifications with background structure $\mathcal{M}$".

2.2.3 Arithmetical Structures

The background structure of interest here is the arithmetical structure, since many SMT solvers have theories which are optimized for solving arithmetic problems. (This structure is the same as that used in [47].) In addition to standard arithmetical operators, it has a collection of *multiset operations*, including max, min, sum and product. For any set R, $fm(R)$ denotes the class of all *finite multisets* over R. Any function $f\colon U \to U$ defines a multiset $mult(f) = \{\!\{f(a) \ : \ a \in U\}\!\}$ over U, the domain of $\mathcal{A}$. A multiset operation (or aggregate) is a function $\Gamma : fm(U) \to U$.

Definition 2.2.4. The *Arithmetical structure* $\mathcal{N}$ is a structure containing at least

$$(\mathbb{N}; 0, 1, \chi, <, +, \cdot, min, max, \Sigma, \Pi),$$

with domain $\mathbb{N}$, the natural numbers, and where min, max, Σ, and Π are multi-set operations and $\chi[\phi](\bar{x})$ is the characteristic function. Other functions, predicates, and multi-set operations may be included, provided every function and relation of $\mathcal{N}$ is polytime computable.

The arithmetical structure $\mathcal{N}$ is one possible for arithmetic in embedded model expansion, but there are endless other possibilities. Any background structure may be used, depending upon the needs of the problem to be specified. It can be useful to use a background vocabulary that contains only predicates, to avoid the difficulties that come with using automated solvers on formulas with functions.

2.2.4 Well-Formed Terms

While the definition of $\mathrm{GGF}_k(\varepsilon)$ *formulas* is applicable to an arbitrary background structures, the definition of well-formed *terms* is specific to the vocabulary ν of the arithmetical structure $\mathcal{N}$. Thus, the logic itself is parameterized with that vocabulary and should be denoted $\mathrm{GGF}_k(\varepsilon, \nu)$. However, since we focus on the arithmetical structure, we will use the simpler notation $\mathrm{GGF}_k(\varepsilon)$.

The structure $\mathcal{N}$ contains multi-set functions. Classical logic does not have terms to denote such objects, so the syntax must be extended as follows. As usual, $\phi(\bar{x})$ denotes that $\bar{x}$ are the free variables of ϕ.

Definition 2.2.5 (well-formed terms)**.** Let τ be the vocabulary $\sigma \cup \nu \cup \varepsilon$ and V a countable set of variables. The set of well-formed terms is the closure of the sets of variables V and constants of τ under the following operations:

1. If f is a τ-function of arity n, other than a multiset operation or the characteristic function, and $\bar{t}$ is a tuple of terms of length n then $f(\bar{t})$ is a term.
2. If Γ is a multiset operation of ν, $f(\bar{x}, \bar{y})$ a term, and $\phi(\bar{x}, \bar{y})$ a τ-formula in which $\bar{x}$ is guarded, then $\Gamma_{\bar{x}}(f(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y}))$, is a term with free variables $\bar{y}$.
3. If $\phi(\bar{x})$ is a τ-formula such that $\exists \bar{x} \phi(\bar{x})$ is a k-guarded formula, then $\chi[\phi]$ is a term with free variables $\bar{x}$.

Multiset operations (case 2) act much like quantifiers, binding the variables $\bar{x}$. Notice that the free variables $\bar{y}$ in ϕ, within a multiset operation term, need not be guarded within ϕ. Their guards are in the formula where the term appears.

2.2.5 Semantics of multiset terms

Let $G(\bar{y})$ be the multiset term $\Gamma_{\bar{x}}(f(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y}))$. The interpretation of $G(\bar{y})$ on τ-structure $\mathcal{D}$ with valuation $\bar{b}$ for $\bar{y}$ is

$$G^{\mathcal{D}}(\bar{b}) = \Gamma\{\!\{ f^{\mathcal{D}}(\bar{a}, \bar{b}) : \mathcal{D} \models \phi[\bar{a}, \bar{b}] \}\!\}. \qquad (1)$$

As usual, $\mathcal{A} \models \phi[\bar{a}]$ means that formula $\phi(\bar{x})$, is true in structure $\mathcal{A}$ when the free variables $\bar{x}$ are taken to denote domain elements $\bar{a}$.

The index $\bar{x}$ in the term $\Gamma_{\bar{x}}$ is not needed in the semantic definition (1) – think e.g. of Γ being summation (Σ). For readability, we may omit ϕ when true and write $\Gamma_{\bar{x}}(f(\bar{x}, \bar{y}))$; omit free variables and write $\Gamma_{\bar{x}}(f : \phi)$.

The interpretation of the characteristic function $\chi[\phi](\bar{x})$ on τ-structure $\mathcal{D}$ with valuation $\bar{a}$ for $\bar{x}$ is: $\chi[\phi]^{\mathcal{D}}(\bar{a}) = 1$ if $\mathcal{D} \models \phi[\bar{a}]$ and 0 otherwise. We may write $\Gamma_{\bar{x}}(f \times \chi[\phi])$ instead of $\Gamma_{\bar{x}}(f : \phi)$ when Γ is invariant under multiple occurrences of 0 in the multiset (i.e., $\Gamma(S) = \Gamma(S \cup \{\!\{0, \ldots, 0\}\!\})$), as is the case for Σ and max on $\mathcal{N}$.

Definition 2.2.6. An *embedded $GGF_k(\varepsilon)$ MX specification with secondary structure* $\mathcal{N}$ is a set of $\mathrm{GGF}_k(\varepsilon)$ sentences over $\sigma \cup \varepsilon \cup \nu$, with terms as in Definition 2.2.5, and the secondary ν-structure is the arithmetical structure of Definition 2.2.4.

In our presentation here, all elements of the active domain are drawn from the arithmetical background structure. All results of the paper generalize to the multi-sorted case, including the case where some domains are not ordered.

The following are examples of embedded MX specifications with secondary structure $\mathcal{N}$, for search versions of two common optimization problems.

Example 2.2.3. **KNAPSACK:** The instance vocabulary is $\{O, w, v, b_v, b_w\}$, where O is the set of objects; w is the weight function; v is the value function; b_w is the weight bound; and b_v is the value bound. The expansion vocabulary is $\{O'\}$, where O' is the set of selected objects.

Upper guard axiom:

$$\forall x(O'(x) \supset O(x)).$$

The axioms are

$$\Sigma_x(w(x) \ : \ O(x) \wedge O'(x)) \leq b_w$$

and

$$b_v \leq \Sigma_x(v(x) \ : \ O(x) \wedge O'(x)),$$

where $t_1 \leq t_2$ is an abbreviation for $t_1 < t_2 \vee t_1 = t_2$. The lower guard for $O(x) \wedge O'(x)$ is $O(x)$.

Example 2.2.4. **MACHINE SCHEDULING PROBLEM [54]:** We must assign jobs to machines to satisfy constraints on release and due dates and a cost bound. The instance lists jobs, machines, possible start times, the release date and due date for each job, the cost and duration for running each job on each machine, and the cost bound. The instance vocabulary, σ, consists of: $Job(j)$, the set of jobs to be scheduled; $Machine(m)$, the set of machines to perform jobs; $Time(t)$, all possible starting times; $ReleaseDate(j)$, a release date for each job; $DueDate(j)$, a due date for each job; $Cost(j, m)$, cost of doing job j on machine m; $Duration(j, m)$, the duration of executing j on m; and c, the cost bound. The active domain consists of all time points, costs, due and release dates, durations, jobs and machines. The expansion vocabulary consists of two functions: $Assignment(j)$ maps jobs to machines and $StartTime(j)$ maps jobs to start times.

Upper guard axioms:
$\forall j \forall m\ (Assignment(j) = m \supset Machine(m) \wedge Job(j))$
$\forall j \forall t\ (StartTime(j) = t \supset Time(t) \wedge Job(j))$

Axioms:
$\Sigma_j(Cost(j, Assignment(j))\ :\ Job(j)) \leq c$
$\forall j(Job(j) \supset StartTime(j) \geq ReleaseDate(j))$
$\forall j(Job(j) \supset StartTime(j) + Duration(j) \leq DueDate(j))$
$\forall t\ (Time(t) \supset (\forall m\ (Machine(m) \supset max_j(count_j(\psi(j, m, t))) = 1))$.

In the last axiom, which specifies that at most one job is on a machine at a time, $count_j(\psi(j, m, t))$ is an abbreviation for $\Sigma_j(\chi[\psi(j, m, t)])$, and ψ defines the set of jobs being executed on machine m at time t, that is, $\psi(j, m, t)$ is:

$$\begin{aligned} &Job(j) \wedge Assignment(j) = m \wedge\ Time(StartTime(j)) \\ &\wedge\ StartTime(j) \leq t \\ &\wedge\ t < StartTime(j) + Duration(j, Assignment(j)), \end{aligned}$$

It is easy to see that all axioms are in $\mathrm{GGF}_k(\varepsilon)$.

An optimization version would include the objective function:

$$minimizing : \Sigma_j(Cost(j, Assignment(j)) : Job(j)).$$

2.3 Grounding

In this section we will describe grounding, the process of constructing a ground formula from a model expansion problem, so that it can be compared to partial grounding which will be described later.

The first step in automated solving of model expansion problems is to use the instance structure to construct a variable-free formula that is equivalent to the given formula. The equivalence that we need is called *model preservation*, meaning that the set of model expansions is equal to the set of models of the ground formula. The authors of [98] define grounding of model expansion problems and describes an algorithm for constructing groundings.

We can also remove occurrences of σ and ν vocabulary by using the instance structure and background structure. Such a simplification is called *reduction* and it can greatly

shorten the formula as well as allowing a solver to solve the formula without knowing about arithmetic. The formula which is produced from the first step of solving is called a *reduced grounding* when it has neither variables nor instance structure vocabulary. With little additional manipulation, a reduced grounding can be solved by a SAT solver.

In order to define *reduced grounding*, some notation must be explained. For a set A, the notation $\tilde{A}$ represents a vocabulary with exactly one constant for each element of A. For structure $\mathcal{B}$, the notation $(\mathcal{B}, \tilde{A}^{\mathcal{B}})$ represents a structure that is the same as $\mathcal{B}$ but defines the vocabulary $\tilde{A}$ by the corresponding elements of A.

Definition 2.3.1 (reduced grounding for MX)**.** From [98], a formula ψ is a reduced grounding of a $(\sigma \cup \varepsilon)$-formula ϕ over a σ-structure $\mathcal{A}$ with domain A if

1. ψ is a ground formula over $\varepsilon \cup \tilde{A}$.
2. For every expansion structure $\mathcal{B}$ of $\mathcal{A}$ over $vocab(\phi)$, $\mathcal{B} \models \psi$ iff $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \phi$.

This definition was created for [98] which dealt with model expansion, and did not include embedded model expansion as we will use here, but the definition of grounding for model expansion works just as well for embedded model expansion if we consider the background structure to be part of the instance structure. In other words, we can modify the definition as follows.

Definition 2.3.2 (reduced grounding for embedded MX)**.** A formula ψ is a reduced grounding of a $(\sigma \cup \nu \cup \varepsilon)$-formula ϕ over a σ-structure $\mathcal{A}$ embedded in ν-structure $\mathcal{M}$ with domain A if

1. ψ is a ground formula over $\varepsilon \cup \tilde{A}$.
2. For every expansion structure $\mathcal{B}$ of $(\mathcal{A}, \mathcal{M})$ over $\sigma \cup \nu \cup \varepsilon$, $\mathcal{B} \models \phi$ iff $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \psi$.

The version for embedded MX is the same as the version for MX, except with $(\mathcal{A}, \mathcal{M})$ in place of $\mathcal{A}$, and $\sigma \cup \nu$ in place of σ. This is because the distinction between background vocabulary and instance vocabulary is meaningless for the definition of grounding, even though it is quite useful for other purposes.

The second condition in the definition is the model preservation property. It is necessary because it ensures that the resulting formula ψ represents the same problem as the original formula ϕ. As long as the model preservation property holds, we are free to transform the formula in any way we feel might make the problem easier to solve, including grounding.

The use of lower guards reduces the task of grounding a formula. Take for example, $\forall x(P(x) \supset E(x))$, with P a predicate of the instance structure that is serving as a guard. In this case, we need only consider assignments for x that are in P which may be much smaller than the instance domain. For this reason, we will assume that the formulas that are to be ground are guarded.

2.4 Grounding with Extended Relational Algebra

In this section we will describe grounding using extended relational algebra as it was introduced in [98].

In order to construct a reduced grounding for first-order logic efficiently, one can take advantage of relational algebra algorithms as described in [98]. Starting at the atoms of the formula, construct relations to represent the instance structure interpretation for each atom, including an empty table for uninterpreted atoms, then use relational algebra operations for each connective of the formula until you have constructed a relation for the formula. In order to produce a formula as the result, the relations must be *extended* by an extra column that holds the formula represented by each row, and the relational algebra operations must be modified to keep the formula correct.

Definition 2.4.1 (Extended X-Relation)**.** Let X be a tuple of variables and A be a domain. An *extended X-relation over A* is a set R of pairs (γ, ψ) such that $\gamma : X \to A$, ψ is a formula, and for all γ_1 and γ_2, if $(\gamma_1, \psi) \in R$ and $(\gamma_2, \psi) \in R$, then $\gamma_1 = \gamma_2$.

In this grounding procedure, an extended relation R is used to represent the grounding of a formula ϕ with the free variables of ϕ being the columns of R, so R contains a ground formula for each variable assignment. When R has no row matching a variable assignment, then for that variable assignment ϕ must ground to $false$. To maintain this property, all relational algebra operations must be redefined to include manipulation of the formula parts of the relations.

To understand how the relational algebra operations are defined for extended relations, think of the formula for each row as a precondition. For row (γ, ψ) with columns γ and formula ψ, γ is to be treated as absent unless ψ is true. For example, if extended relation R_1 contains (γ, ψ_1) and extended relation R_2 contains (γ, ψ_2), then we must include $(\gamma, \psi_1 \vee \psi_2)$ in $R_1 \cup R_2$ to represent the two ways which γ could be in $R_1 \cup R_2$. Similarly, intersection

and join operations may result in conjunctions of formulas when two preconditions must both be satisfied before a row is included in the result.

For an instance atom, the extended relation is the relation provided by the instance structure extended with *true* for every row. For an expansion atom, the extended relation is every possible variable assignment extended by the expansion atom with variables substituted according to the assignment. For a background atom, a relation is constructed by selecting the rows of the relation formed from the guards that satisfy the background atom.

For a conjunction, the procedure can be applied recursively followed by a join, and for a disjunction the answer would be the union of the extended relations. Quantifiers and negations can be grounded with slightly more elaborate procedures.

The notation $\delta_R(\gamma)$ will be used to represent the formula associated with row γ of extended relation R.

Definition 2.4.2 (δ_R)**.** Let R be an extended X-relation over A with $k = |X|$, then δ_R is a function from k-tuples of elements of A to formulas such that:

- for all γ such that $(\gamma, \psi) \in R$, $\delta_R(\gamma) = \psi$;
- for all γ where there does not exist ψ such that $(\gamma, \psi) \in R$, $\delta_R(\gamma) = false$.

Definition 2.4.3 (Extended Relational Algebra)**.** Let R be an extended X-relation and S an extended Y-relation, both over domain A.

- $\neg R$ is the extended X-relation $\neg\mathcal{R} = \{(\gamma, \psi) \mid \gamma : X \to A, \delta_R(\gamma) \neq \top, \text{ and } \psi = \neg\delta_R(\gamma)\}$
- $R \bowtie S$ is the extended $X \cup Y$-relation $R \bowtie S = \{(\gamma, \psi) \mid \gamma : X \cup Y \to A, \gamma|_X \in R, \gamma|_Y \in S, \text{ and } \psi = \delta_R(\gamma|_X) \wedge \delta_S(\gamma|_Y)\}$;
- $R \bowtie^c S$ is the extended $X \cup Y$-relation $R \bowtie^c S = \{(\gamma, \psi) \mid \gamma : X \cup Y \to A, \gamma|_X \in R, \gamma|_Y \in S, \text{ and } \psi = \delta_R(\gamma|_X) \wedge \neg\delta_S(\gamma|_Y)\}$;
- $R \cup S$ is the extended $X \cup Y$-relation $R \cup S = \{(\gamma, \psi) \mid \gamma|_X \in R \text{ or } \gamma|_Y \in S, \text{ and } \psi = \delta_R(\gamma|_X) \vee \delta_S(\gamma|_Y)\}$.
- For $Z \subseteq X$, the *Z-projection* of R, denoted by $\pi_Z(R)$, is the extended Z-relation $\{(\gamma', \psi) \mid \gamma' = \gamma|_Z \text{ for some } \gamma \in R \text{ and } \psi = \bigvee_{\{\gamma \in R \mid \gamma' = \gamma|_Z\}} \delta_R(\gamma)\}$.

- For $Z \subseteq X$, the *Z-quotient* of R, denoted by $d_Z(R)$, is the extended Z-relation $\{(\gamma', \psi) \mid \forall\gamma(\gamma : X \to A \wedge \gamma|_Z = \gamma' \supset \gamma \in R)$, and $\psi = \bigwedge_{\{\gamma \in R|\gamma' = \gamma|_Z\}} \delta_R(\gamma)\}$.
- For $Z \subseteq X$ and T a Z-relation, the *T-quotient* of R, denoted by $d_T(R)$, is the extended $(X \setminus Z)$-relation $\{(\gamma', \psi) \mid \forall\gamma(\gamma \in T \supset \exists\gamma''(\gamma'' \in R \wedge \gamma''|_Z = \gamma \wedge \gamma''|_{(X\setminus Z)} = \gamma'))$, and $\psi = \bigwedge_{\{\gamma \in R|\gamma' = \gamma|_{(X\setminus Z)}\}} \delta_R(\gamma)\}$.

Using the above definitions for relational algebra operations, grounding for non-embedded model expansion can be described as in Figure 2.1, where $\texttt{Gnd}'(\mathcal{A}, \phi)$ is a grounding of ϕ over $\mathcal{A}$. For atom ϕ and structure $\mathcal{A}$, with G being the predicate of ϕ, the notation $\phi(\mathcal{A})$ means the table $G^{\mathcal{A}}$ with each column labeled by the corresponding argument of ϕ. It is assumed that functions do not appear in the formula to complicate $\phi(\mathcal{A})$. For example, let $G^{\mathcal{A}} = \{(1,2),(3,4),(5,6)\}$ and $\phi = G(x,y)$, then

$$\phi(\mathcal{A}) = \begin{array}{cc} x & y \\ \hline 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{array}$$

Procedure $\texttt{Gnd}'(\mathcal{A}, \exists\bar{y}(G_1 \wedge \ldots \wedge G_m \wedge \phi)) = \texttt{Gnd}(\mathcal{A}, \mathcal{R}, \phi')$ where $\mathcal{R} = G_1(\mathcal{A}) \bowtie \ldots \bowtie G_m(\mathcal{A})$ and ϕ' is the result of pushing the negations of ϕ inward to be in front of only existential quantifiers or atoms with no universal quantifiers.
Procedure $\texttt{Gnd}(\mathcal{A}, \mathcal{R}, \phi)$

1. If ϕ is a positive σ-literal, then $\texttt{Gnd}(\mathcal{A}, \mathcal{R}, \phi) = \mathcal{R} \bowtie \phi(\mathcal{A})$;
2. If ϕ is a negative σ-literal $\neg\psi$, then $\texttt{Gnd}(\mathcal{A}, \mathcal{R}, \phi) = \mathcal{R} \bowtie^c \psi(\mathcal{A})$;
3. If ϕ is a ε-literal, then $\texttt{Gnd}(\mathcal{A}, \mathcal{R}, \phi) = \{(\gamma, \phi[\gamma] \mid \gamma \in \mathcal{R}\}$;
4. $\texttt{Gnd}(\mathcal{A}, \mathcal{R}, \phi \wedge \psi) = \texttt{Gnd}(\mathcal{A}, \mathcal{R}, \phi) \bowtie \texttt{Gnd}(\mathcal{A}, \mathcal{R}, \psi)$;
5. $\texttt{Gnd}(\mathcal{A}, \mathcal{R}, \phi \vee \psi) = \texttt{Gnd}(\mathcal{A}, \mathcal{R}, \phi) \cup \texttt{Gnd}(\mathcal{A}, \mathcal{R}, \psi)$;
6. $\texttt{Gnd}(\mathcal{A}, \mathcal{R}, \exists\bar{y}\phi) = \mathcal{R} \bowtie \texttt{Gnd}'(\mathcal{A}, \exists\bar{y}\phi)$;
7. $\texttt{Gnd}(\mathcal{A}, \mathcal{R}, \neg\exists\bar{y}\phi) = \mathcal{R} \bowtie^c \texttt{Gnd}'(\mathcal{A}, \exists\bar{y}\phi)$.

Figure 2.1: Relational Algebra Grounding for (non-embedded) MX as described in [98]

2.5 Satisfiability Modulo Theories

In this section we will describe Satisfiability Modulo Theories, a problem that is like satisfiability but with the assistance of a fixed theory.

Satisfiability Modulo Theories [103] problems in general are first-order logic satisfiability problems except that one or more theories are supplied in addition to the formula to be solved. Solving algorithms are commonly designed with the theories as given.

Commonly, SMT solvers will only accept a fragment of first-order logic, such as the quantifier-free fragment. Arbitrary atomic constants and term constants are usually allowed, but functions and predicates are often restricted to those that the theory deals with, since satisfiability with arbitrary functions and predicates is a harder problem. This sort of SMT solver is like a propositional SAT solver where some propositional variables have been replaced by first-order atoms containing predicates and functions from a list of specially built-in predicates and functions.

Theories often include arithmetic of various kinds because arithmetic is a very convenient abstraction and it can be awkward to define a propositional logic formula by hand to solve arithmetic problems. Other theories can represent data structures such as arrays and bit vectors. When a solver is designed for a certain theory, a fragment logic is also chosen to ensure that the solver can run efficiently, and so solvers are categorized by logic-theory pairs such as `QF_LIA` which stands for quantifier-free linear integer arithmetic. Sometimes the theory is even empty and the solver is doing ordinary satisfiability checking on a fragment of first-order logic, such as in `QF_UF` where uninterpreted functions are allowed. The term *logic* is used to refer to the logic-theory pair that is used in the specification of a problem.

There is an issue about terminology for SMT formulas such as $x + y < 2$. The solver must assign values to x and y, but it is not clear whether x and y are variables or constants. Intuitively they are variables and they are often called variables when explaining a logic like `QF_LIA`. It is useful for clarity to separate x and y as variables from constants like 2 and other numerals. Even so, if this problem were first-order satisfiability, then x and y would be called constants so that satisfiability could be proven by a model instead of a variable assignment. Even [5] is not consistent, sometimes calling the constants variables. For this paper, we will choose to call x and y variables and consider SMT solving to be a problem of finding a variable assignment.

Because most logics used in SMT are small fragments of first-order logic, it is easiest to

list the parts of first-order logic that are allowed rather than describing the restrictions. In the following examples all binary connectives and negation are always allowed, but quantifiers are not allowed unless explicitly stated. The following are examples of logics that SMT solvers support [5]:

`QF_LIA` Quantifier-free linear integer arithmetic uses the theory of integer arithmetic. The following are allowed: propositional variables, term variables, linear arithmetic terms, and the following predicates: $<$, $\leq$, $>$, $\geq$, $=$. For example: $3x < y - 2 \wedge y < 10$.

`QF_LRA` Quantifier-free linear real arithmetic is similar to `QF_LIA` but using the theory of real arithmetic in place of the theory of integer arithmetic. For example: $y + 2 < 3/x \wedge y > 2.3$.

`QF_NIA` Quantifier-free integer arithmetic uses the theory of integer arithmetic. The following are allowed: propositional variables, term variables, arithmetic functions, and the following predicates: $<$, $\leq$, $>$, $\geq$, $=$. For example: $x > 2 \wedge y > 2 \wedge xy < 15$.

`QF_RDL` Quantifier-free difference logic over reals uses the theory of reals. The following are allowed: propositional variables, and atoms of the form $x - y \leq c$, $x - y \geq c$, $x - y < c$, $x - y > c$, $x - y = c$, and $x < y$, where x and y are variables and c is a numeral constant.

`LRA` Linear real arithmetic uses the theory of real arithmetic. The following are allowed: existential and universal quantifiers, propositional variables, term variables, linear arithmetic terms, and the following predicates: $<$, $\leq$, $>$, $\geq$, $=$. For example: $\forall x(x < 1 \vee x > 10 \vee xy < 100)$.

`QF_BV` Quantifier-free bit vectors logic is the quantifier-free fragment of first order logic with the theory of fixed-size bit vectors. The theory of fixed-size bit vectors includes the following functions:

$concat(x, y)$ This term is the concatenation of bit vector y to the end of bit vector x.

$extract(i, j, x)$ This term is the bit vector that represents the bits from index i to index j in bit vector x.

$bvnot(x)$ This term is bit vector x with every 1 bit replaced by 0 and every 0 bit replaced by 1.

$bvand(x, y)$ This term is the bitwise *and* of the two given bit vectors.

$bvor(x, y)$ This term is the bitwise *or* of the two given bit vectors.

There are also $bvadd(x, y)$, $bvmul(x, y)$, $bvudiv(x, y)$, $bvurem(x, y)$, which treat bit vectors as the natural numbers that they represent in binary notation and perform the arithmetic operations that match their names. The term $bvneg(x)$ represents the 2's complement negation of bit vector x. This is not a complete list. The functions of the theory of fixed-size bit vectors are allowed, as are propositional variables, term variables, = atoms, and bit vector constants.

`QF_UF` Quantifier-free uninterpreted functions uses no theory. All unquantified formulas are allowed. The SMT solver must find an interpretation for every predicate and function in the formula.

The SMT-LIB Standard is a standardized notation for representing SMT problems that allows many tools to work upon the same inputs. To participate in contests, an SMT solver must support the SMT-LIB Standard in addition to any other input formats it accepts, so the following is an outline of the important features of the SMT-LIB Standard based on [6].

An SMT problem in the notation of SMT-LIB Standard is a series of commands. The most important commands are `(set-logic QF_LIA)` which puts the solver into a mode to solve problems in the specified logic, and `(assert` ϕ`)` which causes the solver to add the specified formula ϕ to the axiomatization to be solved. In this syntax, formulas are considered terms and predicates are considered boolean functions, so `assert` interprets its parameter as a term that must be evaluated as *true*.

The terms that represent first-order logic connectives are written `(and` $T_1 \ldots T_n$`)`, `(or` $T_1 \ldots T_n$`)`, `(not` T`)`, `(forall ((`x_1 S_1`) ... (`x_n S_n`))` T`)`, `(exists ((`x_1 S_1`) ... (`x_n S_n`))` T`)`, where T and T_1 through T_n are terms, x_1 through x_n are variables, and S_1 through S_n are *sorts* which represent the universe for the quantification of each variable, such as `Int`, `Real`, and `Bool`. In the case of `QF_LIA`, the quantifiers `forall` and `exists` are forbidden.

The command `(declare-fun` F `(`$S_1 \ldots S_n$`)` S`)` specifies that a function F will appear in the formulas with arguments of sorts S_1 through S_n in that order, and with a value of sort S. For the purposes of quantifier-free linear integer arithmetic, we will use the `declare-fun` command in this way: `(declare-fun` x `() Int)`, for some variable x. This indicates that x will be a integer variable that needs to be assigned.

The command `(check-sat)` takes no parameters and simply causes the solver to solve the problem as defined by previous commands. In order to get the solver to report the variable assignment used, the `(get-value (`T_1 `...`T_n`))` command is used, which reports the value of each term it is given.

Chapter 3

Partial Grounding

In this chapter we will define partial grounding and provide two algorithms to construct partial groundings, given a guarded first-order formula and an instance structure.

For example with an instance with universe $\{1, 2, 3, 4\}$, the formula $\exists x \forall y P(x, y)$ could be partially grounded as $P(x, 1) \wedge P(x, 2) \wedge P(x, 3) \wedge P(x, 4)$. For another example with a more difficult arrangement of quantifiers, $\forall x \exists y P(x, y)$ could be partially grounded as $P(1, y_1) \wedge P(2, y_2) \wedge P(3, y_3) \wedge P(4, y_4)$. These examples illustrate how the algorithms that will be defined later construct partial groundings.

In order to deal with variables conveniently we will assume that for any domain A, $\tilde{A}$ is a set of constant symbols that represent all the elements of A, with each element of A represented by exactly one element of $\tilde{A}$. The notation $\mathcal{A} \models \phi$ means that there exists a variable assignment s for the free variables of ϕ such that $\mathcal{A} \models \phi[s]$, and if s is not a partial variable assignment for ϕ, then $\mathcal{A} \models \phi[s]$ means that there exists a total variable assignment t that is an extension of s such that $\mathcal{A} \models \phi[t]$. The notation $\phi[\![s]\!]$ means formula ϕ with the uniform variable substitution specified by function s where variable x is replaced by a constant symbol from $\tilde{A}$ representing $s(x)$.

Definition 3.0.1 (partial grounding for MX)**.** Formula ψ is a *partial grounding* of formula ϕ over structure $\mathcal{A}$ with domain A iff

1. ψ is a quantifier-free formula. In other words, all the first-order variables are free.
2. ψ has vocabulary $vocab(\phi) \cup \tilde{A}$.
3. For every structure $\mathcal{B}$ that is an expansion of $\mathcal{A}$ to $vocab(\phi)$ and for every variable assignment s with domain $free(\phi)$, $\mathcal{B} \models \phi[s]$ iff $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \psi[s]$, where $(\mathcal{B}, \tilde{A}^{\mathcal{B}})$ is the structure obtained by expanding $\mathcal{B}$ by the interpretations of the $\tilde{A}$ with the corresponding elements of A.

While s is a total variable assignment for ϕ in the above definition, $\psi[\![s]\!]$ may still contain some free variables, as in the example for $\forall x \exists y P(x, y)$. The examples given above satisfy this definition of partial grounding. Notice that a partial grounding may contain variables which are not in the original formula.

As an example, take $\phi = \forall x \exists y P(x, y)$, $\psi = P(1, y_1) \wedge P(2, y_2) \wedge P(3, y_3) \wedge P(4, y_4)$, and $\mathcal{A} = (\{1, 2, 3, 4\})$. The first two conditions of partial grounding are clearly satisfied. The third condition requires us to examine every possible expansion $\mathcal{B}$ of $\mathcal{A}$ to $vocab(\phi)$, meaning every possible interpretation of P in the universe of $\{1, 2, 3, 4\}$, and every possible variable assignment s with domain $free(\phi)$. Since ϕ has no free variables, s is empty. If we assume that $\mathcal{B} \models \phi$, then $P^{\mathcal{B}}$ must contain tuples $\{(1, y_1), (2, y_2), (3, y_3), (4, y_4)\}$ for some values of $y_1, \ldots, y_4$. The structure $(\mathcal{B}, \tilde{A}^{\mathcal{B}})$ contains the same interpretation for P, plus interpretations for the numeral constants of $\tilde{A} = \{1, 2, 3, 4\}$. Therefore, for some s', $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \psi[s']$. If we assume $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \psi[s']$, then s' can be used to provide a variable assignment for y for each assignment for x, with $y = s'(y_1)$ for $x = 1$, $y = s'(y_2)$ for $x = 2$, and so on. Therefore, $\mathcal{B} \models \phi$.

Just as with grounding, partial grounding for embedded MX is the same as partial grounding for MX, as long as the background structure is taken together with the instance structure instead of treating them as separate.

The formula $\exists \bar{x} \alpha(\bar{x}, \bar{y})$ cannot by definition appear as part of a partially ground formula ψ, but it is convenient to use it as a notation to represent $\alpha(\bar{z}, \bar{y})$ where $\bar{z}$ is a tuple of variables that occur nowhere else in ψ. This allows our algorithms to avoid the details of variable renaming that may be necessary for some partial groundings. The notation $\eta(\phi)$ for formula ϕ means ϕ with all existential quantifiers removed and all variables renamed as needed.

In this chapter we will introduce two algorithms for constructing partially ground formulas from model expansion problems and prove that the algorithms produce correct results. First will be a simple grounding algorithm, and then a grounding algorithm that uses a modification of the extended relational algebra technique from Section 2.4.

3.1 Naive Grounding

The following grounding procedure requires a formula to be in a guarded fragment (see Section 2.2) and starts by constructing an equivalent formula in negation normal form. This is done so that the resulting formula is still in a guarded fragment, even when a $\forall$ quantifier is replaced by an $\exists$ quantifier or vice-versa. The procedure for this construction is skipped here.

For this algorithm, $\bowtie$ is the join operator with for non-extended relational algebra, since the guard relations that it joins are not extended relations.

Figure 3.1 shows the algorithm for the `Gnd` procedure which performs partial grounding. It assumes that negations only occur in first-order literals, as defined in chapter 2.

Procedure $\texttt{Gnd}(\mathcal{A}, \phi)$

1. If ϕ is a literal then $\texttt{Gnd}(\mathcal{A}, \phi) = \phi$
2. $\texttt{Gnd}(\mathcal{A}, \phi \wedge \psi) = \texttt{Gnd}(\mathcal{A}, \phi) \wedge \texttt{Gnd}(\mathcal{A}, \psi)$
3. $\texttt{Gnd}(\mathcal{A}, \phi \vee \psi) = \texttt{Gnd}(\mathcal{A}, \phi) \vee \texttt{Gnd}(\mathcal{A}, \psi)$
4. If $\phi = \forall \bar{y}((G_1 \wedge \cdots \wedge G_m) \supset \phi')$, then
$$\texttt{Gnd}(\mathcal{A}, \phi) = \bigwedge_{s \in \mathcal{G}} \texttt{Gnd}(\mathcal{A}, \phi')[\![s]\!]$$
$$\text{where } \mathcal{G} = G_1(\mathcal{A}) \bowtie \cdots \bowtie G_m(\mathcal{A})$$
5. If $\phi = \exists \bar{y} \phi'$, then
$$\texttt{Gnd}(\mathcal{A}, \phi) = \exists \bar{y} \texttt{Gnd}(\mathcal{A}, \phi').$$

Figure 3.1: Naive partial grounding

We now prove that the algorithm of section 3.1 produces a partial grounding.

Proposition 3.1.1 (Correctness of the naive algorithm)**.** *Let ϕ be a $\sigma \cup \varepsilon$-formula in negation*

normal form. Let $\mathcal{A}$ be a σ-structure. The formula $\eta(\texttt{Gnd}(\mathcal{A}, \phi))$ is a partial grounding of ϕ over $\mathcal{A}$.

Proof. We can prove the proposition inductively over the structure of the formula $\texttt{Gnd}(\mathcal{A}, \phi)$.

- Let α be a literal. $\eta(\texttt{Gnd}(\mathcal{A}, \alpha)) = \alpha$ and α is a partial grounding of itself because α is a literal.

- Let γ be $\alpha \vee \beta$. Let $\gamma' = \texttt{Gnd}(\mathcal{A}, \gamma) = \alpha' \vee \beta'$. Assume for induction that $\eta(\alpha')$ is a partial grounding of α over $\mathcal{A}$ and $\eta(\beta')$ is a partial grounding of β over $\mathcal{A}$. Assume $\mathcal{B}$ is an expansion of $\mathcal{A}$ and s is a variable assignment from $free(\gamma)$ to A.

  - For one direction of 'iff', assume that $\mathcal{B} \models \gamma[s]$. Then $\mathcal{B} \models \alpha[s]$ or $\mathcal{B} \models \beta[s]$. If $\mathcal{B} \models \alpha[s]$, then $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \alpha'[s]$, by definition of partial grounding and similarly for β and β'. Therefore, $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \alpha'[s]$ or $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \beta'[s]$. Therefore, $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \gamma'[s]$.
  - For the other direction, assume that $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \gamma'[s]$. Then $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \alpha'[s]$ or $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \beta'[s]$. If $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \alpha'[s]$ then $\mathcal{B} \models \alpha[s]$ by definition of grounding and similarly for β' and β. Therefore, $\mathcal{B} \models \alpha[s]$ or $\mathcal{B} \models \beta[s]$. Therefore $\mathcal{B} \models \gamma[s]$.

- Let γ be $\alpha \wedge \beta$. Let $\gamma' = \texttt{Gnd}(\mathcal{A}, \gamma) = \alpha' \wedge \beta'$. Assume for induction that $\eta(\alpha')$ is a partial grounding of α over $\mathcal{A}$ and $\eta(\beta')$ is a partial grounding of β over $\mathcal{A}$. Assume $\mathcal{B}$ is an expansion of $\mathcal{A}$ and s is a variable assignment from $free(\gamma)$ to A.

  - For one direction of 'iff', assume that $\mathcal{B} \models \gamma[s]$. Then $\mathcal{B} \models \alpha[s]$ and $\mathcal{B} \models \beta[s]$. If $\mathcal{B} \models \alpha[s]$, then $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \alpha'[s]$, by definition of grounding and similarly for β and β'. Therefore, $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \alpha'[s]$ and $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \beta'[s]$. Therefore, $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \gamma'[s]$.
  - For the other direction, assume that $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \gamma'[s]$. Then $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \alpha'[s]$ and $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \beta'[s]$. If $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \alpha'[s]$ then $\mathcal{B} \models \alpha[s]$ by definition of grounding and similarly for β' and β. Therefore, $\mathcal{B} \models \alpha[s]$ and $\mathcal{B} \models \beta[s]$. Therefore $\mathcal{B} \models \gamma[s]$.

- Let γ be $\forall \bar{y}((G_1 \wedge \cdots \wedge G_m) \supset \alpha)$. Let $\gamma' = \texttt{Gnd}(\mathcal{A}, \gamma)$ where

$$\texttt{Gnd}(\mathcal{A}, \gamma) = \bigwedge_{s \in \mathcal{G}} \alpha'[s]$$

$$\text{where } \mathcal{G} = G_1(\mathcal{A}) \bowtie \cdots \bowtie G_m(\mathcal{A})$$

Assume for induction that $\eta(\alpha')$ is a partial grounding of α over $\mathcal{A}$. The set $\mathcal{G}$ represents all the relevant values that $\bar{y}$ can take. By definition of grounding for all expansion structures $\mathcal{B}$ and for all $s \in \mathcal{G}$, $\mathcal{B} \models \alpha[s]$ iff $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \alpha'[s]$. Therefore, for all expansion structures $\mathcal{B}$ and for all assignments t, $\mathcal{B} \models \gamma[t]$ iff $(\mathcal{B}, \mathcal{A}^{\mathcal{B}}) \models \gamma'[t]$.

- Let γ be $\exists \bar{y} \alpha$. Let $\gamma' = \texttt{Gnd}(\mathcal{A}, \gamma)$ where

$$\texttt{Gnd}(\mathcal{A}, \gamma) = \exists \bar{y} \alpha'$$

Assume for induction that $\eta(\alpha')$ is a partial grounding of α over $\mathcal{A}$. Assume that $\mathcal{B} \models \gamma[s]$. Then $\mathcal{B} \models \alpha[s]$. By induction, $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \alpha'[s]$, and therefore $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \exists \bar{y} \alpha'$. The other direction is proven similarly. This is possible because in Chapter 3 we defined $\mathcal{B} \models \alpha[s]$ to implicitly existentially quantify all unassigned free variables.

□

3.2 Grounding with Extended Relational Algebra

In this section we describe an algorithm that uses the extended relational algebra of Section 2.4. The algorithm is shown in Figure 3.2. The procedure $\texttt{Gnd}(\mathcal{A}, G, X, \phi)$ has four parameters: $\mathcal{A}$, the instance structure, G, the relation formed from the guards, X, the set of free variables that are to be substituted for partial grounding, and ϕ, the formula. For simplicity, this algorithm assumes that there are no functions in the formula and no negations except literals. This restricts background vocabulary to predicates and constants, but a more elaborate algorithm can avoid this restriction.

To show that the procedure is correct, consider the following proposition.

Proposition 3.2.1 (Correctness of the relational algebra algorithm)**.** *Let ϕ be a $\sigma \cup \varepsilon$-formula with negations only in literals. Let $\mathcal{A}$ be a σ-structure. Let G be a relation. The formula $\texttt{Gnd}(\mathcal{A}, G, \phi)$ is an extended relation R such that for all $\gamma \in G$, $\eta(\delta_R(\gamma))$ is a partial grounding of $\phi[\![\gamma]\!]$ over $\mathcal{A}$.*

Proof. The proposition can be proven inductively, using the literals as base cases.

- Since a literal is a partial grounding for itself, the proposition is clearly true for literals.

Procedure $\texttt{Gnd}(\mathcal{A}, G, \phi)$

1. If ϕ is an expansion predicate literal or a background predicate literal then $\texttt{Gnd}(\mathcal{A}, G, \phi)$ is $\{(\gamma, \phi[\![\gamma]\!]) \mid \gamma \in G\}$.

2. If ϕ is an instance predicate literal and G is an X-relation then $\texttt{Gnd}(\mathcal{A}, G, \phi)$ is the set of all $(\gamma, \phi[\![\gamma]\!])$ such that $\gamma \in \pi_X(G \bowtie \phi(\mathcal{A}))$

3. $\texttt{Gnd}(\mathcal{A}, G, \phi \wedge \psi) = \texttt{Gnd}(\mathcal{A}, G, \phi) \bowtie \texttt{Gnd}(\mathcal{A}, G, \psi)$

4. $\texttt{Gnd}(\mathcal{A}, G, \phi \vee \psi) = \texttt{Gnd}(\mathcal{A}, G, \phi) \cup \texttt{Gnd}(\mathcal{A}, G, \psi)$

5. $\texttt{Gnd}(\mathcal{A}, G, \forall \bar{y}((G_1 \wedge \cdots \wedge G_n) \supset \phi)) = d_{G'}(\texttt{Gnd}(\mathcal{A}, G'', \phi))$ where

$$G' = \pi_{\bar{y}}(G_1(\mathcal{A}) \bowtie \cdots \bowtie G_n(\mathcal{A}))$$
$$G'' = G \bowtie G'$$

6. $\texttt{Gnd}(\mathcal{A}, G, \exists \bar{y}\phi)$ is $\{(\gamma, \exists \bar{y}\psi) \mid (\gamma, \psi) \in \texttt{Gnd}(\mathcal{A}, G, \phi)\}$

Figure 3.2: Relational algebra partial grounding

- For $\phi \wedge \psi$, assume the proposition for $\texttt{Gnd}(\mathcal{A}, G, \phi) = R$ and $\texttt{Gnd}(\mathcal{A}, G, \psi) = S$ for induction. Let $\gamma \in G$, then $\eta(\delta_R(\gamma))$ is a partial grounding of $\phi[\![\gamma]\!]$ and $\eta(\delta_S(\gamma))$ is a partial grounding of $\psi[\![\gamma]\!]$. Therefore $\eta(\delta_R(\gamma) \wedge \delta_S(\gamma))$ is a partial grounding of $(\phi \wedge \psi)[\![\gamma]\!]$. Therefore, the proposition is true for $\texttt{Gnd}(\mathcal{A}, G, \phi \wedge \psi) = R \bowtie S$.

- For $\phi \vee \psi$, assume the proposition for $\texttt{Gnd}(\mathcal{A}, G, \phi) = R$ and $\texttt{Gnd}(\mathcal{A}, G, \psi) = S$ for induction. Let $\gamma \in G$, then $\eta(\delta_R(\gamma))$ is a partial grounding of $\phi[\![\gamma]\!]$ and $\eta(\delta_S(\gamma))$ is a partial grounding of $\psi[\![\gamma]\!]$. Therefore $\eta(\delta_R(\gamma) \vee \delta_S(\gamma))$ is a partial grounding of $(\phi \vee \psi)[\![\gamma]\!]$. Therefore, the proposition is true for $\texttt{Gnd}(\mathcal{A}, G, \phi \vee \psi) = R \cup S$.

- For $\forall \bar{y}((G_1 \wedge \cdots \wedge G_n) \supset \phi)$, assume the proposition for $\texttt{Gnd}(\mathcal{A}, G'', \phi) = R$ for induction where $G' = \pi_{\bar{y}}(G_1(\mathcal{A}) \bowtie \cdots \bowtie G_n(\mathcal{A}))$ and $G'' = G \bowtie G'$. Let $d_{G'}(R) = S$. Let $\gamma \in G$, then an extension γ'' of γ is in G'', and $\eta(\delta_R(\gamma''))$ is a partial grounding of $\phi[\![\gamma'']\!]$. By division, $\delta_S(\gamma)$ is the conjunction of $\delta_R(\gamma_0'')$ for all $\gamma_0'' \in R$ where γ_0'' is an extension of γ. Therefore, $\eta(\delta_S(\gamma))$ is a partial grounding of $(\forall \bar{y}((G_1 \wedge \cdots \wedge G_n) \supset \phi))[\![\gamma]\!]$. Therefore, the proposition is true for $\texttt{Gnd}(\mathcal{A}, G, \forall \bar{y}((G_1 \wedge \cdots \wedge G_n) \supset \phi)$.

- For $\exists \bar{y}\phi$, assume the proposition for $\texttt{Gnd}(\mathcal{A}, G, \phi) = R$ for induction. Given that R is a correct extended relation of groundings, the proposition is clear for $\texttt{Gnd}(\mathcal{A}, G, \exists \bar{y}\phi)$

since it is the result of merely adding an existential quantifier.

□

Chapter 4

Solving Model Expansion Using Partial Grounding

In this chapter we describe how to construct an SMT problem from a partial grounding of a model expansion problem.

4.1 Reduction Algorithm

Given a $(\sigma \cup \nu \cup \varepsilon)$-formula ϕ and a $(\sigma \cup \nu)$-structure $\mathcal{A}$, our goal is to find an $(\sigma \cup \nu \cup \varepsilon)$-structure $\mathcal{B}$ which is an expansion of $\mathcal{A}$ such that $\mathcal{B} \models \phi$. In order to use partial grounding to find $\mathcal{B}$, let us split $\mathcal{A}$ into two structures, the instance structure $\mathcal{C}$ of vocabulary σ and the background structure $\mathcal{D}$ of vocabulary ν as described in section 2.2.1. Informally structure $\mathcal{C}$ is an encoding of a problem instance and $\mathcal{D}$ is the arithmetic that we want to use to solve the problem. Instead of being given the structure $\mathcal{D}$, we will use an SMT solver that will produce a variable assignment that allows $\mathcal{D}$ to satisfy a given ν-formula, if such an assignment exists.

The following defines what we need an SMT solver to do. The abilities of actual SMT solvers are described in section 2.5.

Definition 4.1.1 (SMT-solver)**.** For ν-structure $\mathcal{D}$, a total function f from first order formulas to variable assignments is an *$\mathcal{D}$-SMT-solver* iff

1. The domain of f is the set of first-order ν-formulas that contain no universal quantifiers.

2. For any ϕ in the domain of f, if there exists a variable assignment s such that $\mathcal{D} \models \phi[s]$ then $\mathcal{D} \models \phi[f(\phi)]$.

Given a $(\sigma \cup \nu \cup \varepsilon)$-formula ϕ, a σ-structure $\mathcal{C}$, and a $\mathcal{D}$-SMT-solver S for some ν-structure $\mathcal{D}$, we need to find a $(\sigma \cup \nu \cup \varepsilon)$-structure $\mathcal{B}$ which is an expansion of $\mathcal{C}$ and $\mathcal{D}$ such that $\mathcal{B} \models \phi$. Let ψ be a partial grounding of ϕ over $\mathcal{C}$ such that ψ contains no universal quantifiers and no negations except negative literals. We will use ψ, $\mathcal{C}$, and S to construct $\mathcal{B}$ if any such structure exists.

We will make three assumptions that are true of almost all SMT solvers in practice. First, assume that ν contains $=$ so that we can have subformulas of the form $x = a$ where x is a variable required to have the value of constant a. We also assume that the domain of $\mathcal{D}$ contains at least two elements so that we can represent true and false, and ν contains at least one constant which we will call c so that we can have subformulas of the form $x = c$ to indicated that x is a propositional variable that must be true in the resulting structure. We will also assume ν contains a constant for every element in the domain of $\mathcal{C}$ so that new constants introduced in grounding are always in ν.

We start by constructing a ν-formula ψ' from ψ so that we can give ψ' to S. This requires removing all atoms containing vocabulary from ε or σ. Since S is a $\mathcal{D}$-SMT-solver, the domain of S is ν-formulas. We replace each σ atom α with ν-formula β such that for all variable assignments s, $\mathcal{D} \models \beta[s]$ if and only if $\mathcal{C} \models \alpha[s]$. The formula β is constructed from α and $\mathcal{C}$ by using equations.

The handling of σ atoms is straight-forward, so let us illustrate it by an example. If P is a predicate of 3 arguments in σ, and $P^{\mathcal{C}} = \{(1,2,3),(2,2,2),(3,3,3)\}$ then where $P(x,3,a)$ appears in ψ, there will be the following in the ν-formula ψ':

$$(x = 3 \wedge 3 = 2 \wedge a = 3) \vee (x = 2 \wedge 3 = 2 \wedge a = 2) \vee (x = 3 \wedge 3 = 3 \wedge a = 3)$$

Before we finish constructing ψ', we need to convert any ε-literals into ν-formulas. This conversion is very different depending upon whether the ε-literal is positive or negative. For a positive literal α, we create a new variable x_α and construct the ν-equation $x_\alpha = c$. Let D be the set of all positive ε-literals in ψ, then for any negative ε-literal $\neg P(\bar{x})$ in ψ, we construct a conjunction

$$\bigwedge_{P(\bar{y}) \in D} (x_{P(\bar{y})} \neq c \vee \bar{y} \neq \bar{x})$$

The above conjunction says that for every positive expansion literal with predicate P and arguments $\bar{y}$, either that literal is false or $\bar{y}$ does not match the arguments of $\neg P(\bar{x})$. The reason for this is to avoid any inconsistencies when the SMT solver chooses a truth assignment for x_α for each positive ε-literal α.

The goal of this construction for ε is to have the SMT solver assign true or false to every positive ε-literal in ψ, so each distinct positive ε-literal is given a variable. Since the SMT solver does not understand that the variables represent atoms, we need to include the meaning of atoms in ψ'. We do this by replacing each negative ε-literal with a formula that exhaustively tests each variable for a truth value that would make the negative ε-literal false. For example, the formula for $\neg P(1)$ would contain a conjunct representing that the variable for $P(1)$ is assigned false, and another conjunct that say that if $x = 1$, then $P(x)$ must be assigned false. In this way, we encode the meaning of both positive and negative ε-literals in ψ'.

Figure 4.1 shows the procedure for the construction of ψ' more formally, where $\psi' := \texttt{Reduce}(\mathcal{C}, N, \varepsilon, \nu, \alpha)$ and α is ψ in negation normal form. The set N is all positive ε-literals in ψ. For tuples $\bar{t} = (t_1, \ldots, t_n)$ and $\bar{x} = (x_1, \ldots, x_n)$, the notation $\bar{t} = \bar{x}$ is a short form for $t_1 = x_1 \wedge \cdots \wedge t_n = x_n$.

Once we have ψ', we use $S(\psi')$ to construct $\mathcal{B}$. If $\mathcal{D} \not\models \psi'[S(\psi')]$ then $\mathcal{B}$ does not exist. Otherwise, we start constructing $\mathcal{B}$ by assuming it is an expansion of $\mathcal{C}$ and $\mathcal{D}$ so that we only have to provide interpretations for the elements of ε.

We use N and $S(\psi')$ to construct an interpretation for each predicate P in ε as follows:

$$P^{\mathcal{B}} = \{\bar{t}[\![S(\psi')]\!] \mid P(\bar{t}) \in N, S(\psi')(x_{P(\bar{t})}) = c\}$$

This works because the SMT solver provides truth values for each positive ε-literal α in the form of a variable x_α as well as values for each variable in the arguments of α. To construct $\mathcal{B}$, all we have to do is create a structure which agrees with the truth values that the SMT

Procedure $\texttt{Reduce}(\mathcal{A}, N, \varepsilon, \nu, \phi)$

1. $\texttt{Reduce}(\mathcal{A}, N, \varepsilon, \nu, \phi \wedge \psi) = \texttt{Reduce}(\mathcal{A}, N, \varepsilon, \nu, \phi) \wedge \texttt{Reduce}(\mathcal{A}, N, \varepsilon, \nu, \psi)$
2. $\texttt{Reduce}(\mathcal{A}, N, \varepsilon, \nu, \phi \vee \psi) = \texttt{Reduce}(\mathcal{A}, N, \varepsilon, \nu, \phi) \vee \texttt{Reduce}(\mathcal{A}, N, \varepsilon, \nu, \psi)$
3. $\texttt{Reduce}(\mathcal{A}, N, \varepsilon, \nu, \phi) = \phi$ if ϕ is a ν-literal.
4. For any predicate symbol E in ε,

$$\texttt{Reduce}(\mathcal{A}, N, \varepsilon, \nu, \neg E(\bar{t})) = \bigwedge_{E(\bar{y}) \in N} (x_{E(\bar{y})} \neq c \vee \bar{y} \neq \bar{t})$$

$$\texttt{Reduce}(\mathcal{A}, N, \varepsilon, \nu, E(\bar{t})) = (x_{E(\bar{t})} = c)$$

5. For any predicate symbol P in the vocabulary of $\mathcal{A}$ then

$$\texttt{Reduce}(\mathcal{A}, N, \varepsilon, \nu, \neg P(\bar{t})) = \neg \bigvee_{\bar{x} \in P^{\mathcal{A}}} (\bar{t} = \bar{x})$$

$$\texttt{Reduce}(\mathcal{A}, N, \varepsilon, \nu, P(\bar{t})) = \bigvee_{\bar{x} \in P^{\mathcal{A}}} (\bar{t} = \bar{x})$$

Figure 4.1: `Reduce`

solver provides.

Theorem 4.1.2. *Let $\mathcal{C}$ be a σ-structure with domain C and ν is a vocabulary containing true such that σ and ν are disjoint. Let $\mathcal{D}$ be a ν-structure with a domain of size at least 2. Let S be a $\mathcal{D}$-SMT solver. Let ϕ be a $(\sigma \cup \nu \cup \varepsilon)$-formula. Let ψ be a partial grounding of ϕ with respect to $\mathcal{C}$.*

If

- *N is the set of all positive ε-literals in ψ*
- *$\psi' = \texttt{Reduce}(\mathcal{C}, N, \varepsilon, \nu, \psi)$ and ψ' is an ν-formula*
- *$\mathcal{D} \models \psi'[S(\psi')]$*
- *$\mathcal{B}$ is a $(\sigma \cup \nu \cup \varepsilon)$-structure that is an expansion of $\mathcal{C}$ and $\mathcal{D}$ and such that for any predicate $P \in \varepsilon$*

$$P^{\mathcal{B}} = \{\bar{t}[\![S(\psi')]\!] \mid P(\bar{t}) \in N, S(\psi')(x_{P(\bar{t})}) = c\}$$

then $\mathcal{B} \models \phi$.

Proof. Let ψ be a partial grounding of ϕ with respect to $\mathcal{C}$ in negation normal form.

There are six kinds of literals, positive and negative from each of the three vocabularies, σ, ν, ε. For each literal α in ψ, $\texttt{Reduce}(\mathcal{C}, N, \varepsilon, \nu, \psi)$ uniformly substitutes a formula α' in ψ'. We will show that $\mathcal{B} \models \alpha[S(\psi')]$ if $\mathcal{D} \models \alpha'[S(\psi')]$.

- Let α be a σ-literal. From the definition of `Reduce`, α' is a disjunction of equations representing the corresponding relation in $\mathcal{C}$, or the negation of that if α is negative. Assume $\mathcal{D} \models \alpha'[S(\phi')]$. Since variable assignment is the same, clearly $\mathcal{C} \models \alpha[S(\phi')]$, and since $\mathcal{B}$ is an expansion of $\mathcal{C}$, $\mathcal{B} \models \alpha[S(\phi')]$. Similarly, if we assume $\mathcal{B} \models \alpha[S(\phi')]$ then we must have $\mathcal{C} \models \alpha[S(\phi')]$ since α is a σ-literal, and therefore $\mathcal{D} \models \alpha'[S(\phi')]$.

- Let α be a ν-literal. From the definition of `Reduce`, $\alpha' = \alpha$. Since $\mathcal{B}$ is an expansion of $\mathcal{D}$, it is clear that $\mathcal{D} \models \alpha'[S(\psi')]$ iff $\mathcal{B} \models \alpha[S(\psi')]$.

- Let α be a positive ε-literal. From the definition of `Reduce`, $\alpha' = (x_\alpha = c)$ where x_α is a variable. Assume $\mathcal{D} \models \alpha'[S(\psi')]$. The variable assignment $S(\psi')$ must assign c to x_α. Directly from the construction of $\mathcal{B}$, $\mathcal{B} \models \alpha[S(\psi')]$. Similarly, if we assume $\mathcal{B} \models \alpha[S(\psi')]$ then the construction of $\mathcal{B}$ ensures that $S(\psi')(x_\alpha) = c$ and so $\mathcal{D} \models \alpha'[S(\psi')]$.

- Let α be the negative ε-literal $\neg E(\bar{t})$. From the definition of `Reduce`,

$$\alpha' = \bigwedge_{E(\bar{y}) \in N} (x_{E(\bar{y})} \neq c \vee \bar{y} \neq \bar{t})$$

  Assume $\mathcal{D} \models \alpha'[S(\psi')]$. For all $E(\bar{y})$ in D, either $S(\psi')(x_{E(\bar{y})}) \neq c$ or $\bar{y}[S(\psi')] \neq \bar{t}[S(\psi')]$. By the construction of $\mathcal{B}$, the only way $\mathcal{B} \models E(\bar{y})[S(\psi')]$ can be true is if both $S(\psi')(x_{E(\bar{y})}) = c$ and $\bar{y}[S(\psi')] = \bar{t}[S(\psi')]$ for some $E(\bar{y})$ in N, therefore $\mathcal{B} \models \neg E(\bar{y})[S(\psi')]$. If we assume that $\mathcal{B} \models \neg E(\bar{y})[S(\psi')]$ then we have $S(\psi')(x_{E(\bar{y})}) \neq c$ and $\bar{y}[S(\psi')] \neq \bar{t}[S(\psi')]$ for all $E(\bar{y})$ in D and therefore $\mathcal{D} \models \alpha'[S(\psi')]$.

By induction, we can now prove that for any ψ that meets the above conditions, $\mathcal{B} \models \psi[S(\texttt{Reduce}(\mathcal{C}, D, \varepsilon, \nu, \psi))]$. For formulas α and β, where $\alpha' = \texttt{Reduce}(\mathcal{C}, D, \varepsilon, \nu, \alpha)$ and $\beta' = \texttt{Reduce}(\mathcal{C}, N, \varepsilon, \nu, \beta)$, assume $\mathcal{B} \models \alpha[S(\psi')]$ iff $\mathcal{D} \models \alpha'[S(\psi')]$ and $\mathcal{B} \models \beta[S(\psi')]$ iff $\mathcal{D} \models \beta'[S(\psi')]$.

- Let $\gamma = \alpha \vee \beta$. Let $\gamma' = \texttt{Reduce}(\mathcal{C}, D, \varepsilon, \nu, \gamma) = \alpha' \vee \beta'$. Assume $\mathcal{D} \models \gamma'[S(\psi')]$, then either $\mathcal{D} \models \alpha'[S(\psi')]$ or $\mathcal{D} \models \beta'[S(\psi')]$. By assumption, either $\mathcal{B} \models \alpha[S(\psi')]$ or $\mathcal{B} \models \beta[S(\psi')]$ and therefore $\mathcal{B} \models \gamma[S(\psi')]$.

If we assume $\mathcal{B} \models \gamma[S(\psi')]$, then either $\mathcal{B} \models \alpha[S(\psi')]$ or $\mathcal{B} \models \beta[S(\psi')]$ and therefore $\mathcal{D} \models \gamma'[S(\psi')]$.

- Let $\gamma = \alpha \wedge \beta$. Let $\gamma' = \texttt{Reduce}(\mathcal{C}, D, \varepsilon, \nu, \gamma) = \alpha' \wedge \beta'$. Assume $\mathcal{D} \models \gamma'[S(\psi')]$, then $\mathcal{D} \models \alpha'[S(\psi')]$ and $\mathcal{D} \models \beta'[S(\psi')]$. By assumption, $\mathcal{B} \models \alpha[S(\psi')]$ and $\mathcal{B} \models \beta[S(\psi')]$ and therefore $\mathcal{B} \models \gamma[S(\psi')]$.

  If we assume $\mathcal{B} \models \gamma[S(\psi')]$, then $\mathcal{B} \models \alpha[S(\psi')]$ and $\mathcal{B} \models \beta[S(\psi')]$ and therefore $\mathcal{D} \models \gamma'[S(\psi')]$.

- Let $\gamma = \exists x\ \alpha$. Let $\gamma' = \texttt{Reduce}(\mathcal{C}, N, \varepsilon, \nu, \gamma) = \alpha'$. Assume $\mathcal{D} \models \alpha'[S(\psi')]$. By assumption, $\mathcal{B} \models \alpha[S(\psi')]$ and therefore $\mathcal{B} \models \exists x\ \alpha[S(\psi')]$.

□

4.2 Expansion functions as SMT variables

Previous sections have assumed that there are no functions in the formula to simplify the algorithms, either because all other functions were eliminated during the grounding process or because equivalent predicates were used in place of functions. Sometimes it is impractical to write a formula without functions and impossible to eliminate those functions in grounding because they are expansion functions. In that case, it is possible to substitute each function term with a variable term that an SMT solver can accept.

Let f be an ε-function. Let ϕ be a formula. Let N_f be the set of tuples of terms that represent the arguments of each instance of f that appears in ϕ. For every tuple of terms $\bar{x} \in N_f$, let $f_{\bar{x}}$ be a new variable that represents the value of the term $f(\bar{x})$. For every unordered pair $\{\bar{x}, \bar{y}\} \subseteq N_f$, let $\psi_{\bar{x},\bar{y}}$ be the formula $\bar{x} = \bar{y} \supset f_{\bar{x}} = f_{\bar{y}}$, which is a legal SMT formula for ensuring that if the arguments of f are equal then the terms are equal. Since an SMT solver does not know that we are using a set of variables to represent each f, we can explicitly add $\bigwedge_{\{\bar{x},\bar{y}\} \subseteq D_f} \psi_{\bar{x},\bar{y}}$ to ensure that the implicit rules of being a function are followed in any solution found.

For example, the partially ground formula $f(x, y) = g(1) + f(1, 2) \wedge f(0, 0) < g(0)$ would become $f_{x,y} = g_1 + f_{1,2} \wedge f_{0,0} < g_0 \wedge \psi$ where ψ is the explicit condition that f and g are functions as follows:

$$\psi = \wedge \begin{array}{l} (x = 1 \wedge y = 2) \supset f_{x,y} = f_{1,2}) \\ (x = 0 \wedge y = 0) \supset f_{x,y} = f_{0,0}) \end{array}$$

The formula ψ excludes conjuncts that simplify to *true* such as $1 = 0 \supset g_1 = g_0$ and $(1 = 0 \wedge 2 = 0) \supset f_{1,2} = f_{0,0}$, since they are unnecessary.

Chapter 5

The Tool

In this chapter we describe the construction of a software tool for constructing partial groundings from model expansion problems. The first version is described in detail, then we briefly discuss the changes made for the second version in Section 5.6. The most important classes in the object-oriented design are explained and detailed. In Section 5.5, we describe the language that the tool uses to represent model expansion problems.

The following first version of the partial grounding tool is heavily based upon the grounding tool developed by Amir Aavani and Shahab Tasharrofi. While preserving as much as possible of the original software, we made the modifications necessary to create the first version of the partial grounder out of their work.

5.1 BaseFOFormulaNode class

The fundamental class of the grounder's object oriented design is called BaseFOFormulaNode, which is the base class of all node objects which make up a formula. The functions of BaseFOFormulaNode and its subclasses are responsible for all operations upon formulas, and there are few design considerations provided for performing operations in any other way. Therefore, a list of the functions of BaseFOFormulaNode is a good way to get an overview of capabilities of the grounder.

The following is a partial list of the functions of BaseFOFormulaNode.

ToXML() This produces an XML representation of the formula. This is for debugging purposes, to represent most details of the internal structure of the formula, including

things which could not be represented in first order formula notation. Since its only use is debugging, the details of the representation are omitted.

ToString(*Assignment***)** Given an AssignmentCollection object, this renders the formula as a string that resembles standard first order formula notation. The given AssignmentCollection object represents a variable assignment that will be used before rendering.

ToSMT(*Atoms***,** *IsPositive***,** *Assignments***)** Given a PosAtomCollection object, a boolean, and an AssignmentCollection object, this renders the formula in a form that an SMT solver can recognize using the standard contest notation. The PosAtomCollection object represents a list of every instance of expansion predicate atoms appearing in the overall formula, including parent nodes and nodes of entirely different branches. This list has to be created before ToSMT can be called because ToSMT performs a transformation on negative expansion predicates that depends on the list. AssignmentCollection is a variable assignment that allows for variable substitutions.

The IsPositive boolean variable represents whether an even or an odd number of negations contain this node. This is necessary because ToSMT renders positive expansion atoms differently from negative expansion atoms and IsPositive determines which representation is used. The IsPositive parameter makes it unnecessary to have the formula in negation normal form.

CollectPosAtoms(*Atoms***,** *IsPositive***,** *Assignments***)** Given a PosAtomCollection object, a boolean, and an AssignmentCollection object, this searches this node and its children for positive atoms and modifies the given PosAtomCollection object to include everything found. The AssignmentCollection object is given because the BaseFOFormulaNode data structure sometimes assigns values to variables at certain nodes, and those assignments need to be stored with the atoms.

PushNegation(*negative***)** Given a boolean, this constructs a new node tree to represent this formula after it has been transformed to an equivalent negation normal form formula. It works by recursively examining the children of this node and using the given boolean argument to keep track of whether atoms should be positive or negative when they are discovered.

DoGrounding() This constructs a TableCls object that represents the ground form of

this formula. It works by recursively constructing TableCls objects for its children and then using relational algebra operations to create the table for this node.

Intuitively `DoGrounding()` should take parameters to represent the universe and the interpretations for instance predicates, but in this implemetation it requires no parameters because at each node where such information is needed there is a BasicInfo object which contains all needed interpretations. BasicInfo has several subclasses (PredicateSymbolInfo, FunctionInfo, IntegerOperatorInfo, VariableInfo, IntegerContainerInfo), one for each kind of situation where interpretation is needed. In other words, the tree of BaseFOFormulaNode objects represents both the formula and the instance. BasicInfo will be discussed in detail later.

The following is a partial list of the subclasses of BaseFOFormulaNode which are used to represent the various formulas the tool can accept and report.

OperatorFOFormulaNode This is the class of nodes that may have children.

Subclasses:

UnaryOperatorFormulaNode This is the class of objects that represent unary operators, including quantifiers because a quantifier node has only a single child.

- NotOpFOFormulaNode
- QuantifierFOFormulaNode
  - ExistentialQuanFOFormulaNode
  - ForAllQuanFOFormulaNode
  - ConditionalQuanFOFormulaNode
    - GuardedExistentialQuanFOFormulaNode
    - GuardedForAllQuanFOFormulaNode
- RenamingFormulaNode
  - AssignValueToVarFormulaNode

BinaryOperatorFormulaNode This is the class of objects that represent binary operators.

- AndOpFOFormulaNode
- OrOpFOFormulaNode

- ImpliesOpFOFormulaNode
- IffOpFOFormulaNode

GeneralOperatorFormulaNode This is the class of objects that represent formulas with more than two subformulas. Each subclass serves the same purpose as the corresponding BinaryOperatorFormulaNode subclass, but with more arguments.

- GeneralAndOpFOFormulaNode
- GeneralOrOpFOFormulaNode

AtomicFormula This is the class of nodes whose children are TermNodes instead of BaseFOFormulaNodes.

Subclasses:

RelationFOFormulaNode This class brings together a PredicateSymbolInfo object which contains the interpretation of a predicate, with the TermNode arguments which are needed to determine if an atom is true or false.

EqualityFOFormulaNode This is the class of nodes for identity relations.

OrderingFOFormulaNode This is the class of nodes for inequality relations.

TrueFOFormulaNode These nodes represent *true* as a propositional constant.

FalseFOFormulaNode These nodes represent $false$ as a propositional constant.

The RenamingFormulaNode class serves a special purpose in the graph of a formula's nodes. It allows a formula to reuse nodes instead of being represented by a tree. Even if the same node is used in several places, by use of a RenamingFormulaNode, it can have a different interpretation in each place by substituting variables with terms. Each RenamingFormulaNode has a pointer to a VariableInfo that represents a variable to be substituted, while the subclasses specify what sort of term should replace the variable. In fact, AssignValueToVarFormulaNode is the only subclass of RenamingFormulaNode. Each AssignValueToVarFormulaNode has a VariableInfo object and a Value object, and represents that for the child formula of this node every occurrence of that VariableInfo is to be interpreted as the Value.

AssignValueToVarFormulaNode takes the place of term nodes which would be used to represent constant value terms. Wherever a constant with a known value is needed, a variable and a AssignValueToVarFormulaNode can be used.

The AssignmentCollection class of objects is used to keep track of which AssignValueToVarFormulaNodes have been seen while traversing the node graph of a formula. Without this, it would be impossible to determine what a variable represents. For the same reason, when we construct lists of atoms in a formula, we must store (BaseFOFormulaNode, AssignmentCollection) pairs, since a node alone would be meaningless.

5.2 TermNode class

The TermNode class is similar to BaseFOFormulaNode class, but it represents nodes of terms instead of formulas. The following is a list of functions for TermNode.

GetReference() This simply returns a pointer to the BasicInfo object for this term. (Is it assumed that all terms have info? What about aggregates? Investigation needed.)

ToXML() This produces an XML representation of the term.

ToSMT(*Atoms***,** *Assignment***)** Given a PosAtomCollection object and an AssignmentCollection object, this renders the term into a string that an SMT solver would understand.

ToSMTName(*Assignment***)** Given an AssignmentCollection object to provide values for the variables of this node, this produces a legal name in the SMT Standard that will represent this term. It begins with the function name or variable name and if there are any arguments they are surrounded by `'`and separated by `_`. For example, $f(x, y)$ with assigmnent $\{(y, 3)\}$ would be represented as `f'x_3'`. The result is something that a human can recognize as the term it came from, but an SMT solver will see as nothing but a single name.

GetFreeVariables() This produces a list of VariableInfo objects that represent variables that are unquantified.

GetGroundableVariables() This produces a list of VariableInfo objects that represent variables that are to be substituted for values during grounding.

IsEqual(*Node***)** Given a TermNode object, this returns true if this term has the same syntax as the given term.

Evaluate(*Assignments*) Given an AssignmentCollection object, this returns a Value object that represents the interpretation of the term.

The following is a partial list of subclasses for TermNode.

VarSymbTermNode This is a class of TermNodes that have no arguments and contain VariableInfo objects that can be used with AssignmentCollection objects to produce Value objects. Evaluate(*Assignments*) uses the given AssignmentCollection with the contained VariableInfo.

FunctionTermNode This is a class of TermNodes that uses a FunctionInfo object and the evaluations of its children to produce its Value object.

IntTermNode This is the class of nodes that can be involved in integer arithmetic. For all IntTermNodes except for IntegerContainerTermNodes, the children are only allowed to be other IntTermNodes. In addition to Evaluate(x), IntTermNodes have IntegerEvaluate(x) that does the same calculation but returns an `int` instead of an object. Evaluate(*Assignments*) calls IntegerEvaluate(*Assignments*) and constructs a Value object around the resulting `int`.

Subclasses:

IntegerContainerTermNode This class of nodes wraps a TermNode in an IntTermNode to allow variables, functions, and constants to be part of arithmetic. IntegerEvaluate(*Assignments*) calls Evaluate(*Assignments*) and extracts an `int` from the resulting Value object.

IntegerOperatorTermNode This is the class of nodes that represent arithmetic operators with built-in integer arithmetic functions. Each of its subclasses differ only in how they implement IntegerEvaluate(x).

- SumOperatorTermNode
- SubtractOperatorTermNode
- MultiplicationOperatorTermNode

5.3 BasicInfo class

The BasicInfo class is the superclass of all the objects which represent vocabulary with interpretations. The important values stored in every BasicInfo object are a String for the

name of the object and a TypeInfo object which will be explained later. Each subclass adds whatever interpretation is needed for how the subclass is to be used.

The following is a partial list of BasicInfo subclasses.

VariableInfo These objects are held by quantifier nodes and VarSymbTermNodes to keep track of where a variable occurs. One object is used for each variable, and each instance of that variable is a pointer to the object within a VarSymbTermNode. For the purposes of partial grounding, each VariableInfo has a ToBeGround boolean variable which is true if the variable is universally quantified and needs to be substituted in the grounding.

FunctionInfo Each object of this class contains a function interpretation in the form of a DataSet object. Each FunctionTermNode contains a pointer to a FunctionInfo to name the function and so that the formula can be evaluated. For functions where no interpretation is available, the DataSet pointer of the FunctionInfo is null.

IntegerOperatorInfo This subclass of FunctionInfo has additional functions to represent integer arithmetic. GetValue(x) takes a list of Value objects and produces a Value object that is the appropriate result for the particular operator. GetIntegerResult(x) takes a list of `int`s and returns an `int`, for the same purpose as GetValue(x).

- SumOperatorInfo
- SubtractOperatorInfo
- ProductOperatorInfo

PredicateSymbolInfo This class is similar to FunctionInfo, but it serves predicates instead of functions, and RelationFOFormulaNodes instead of FunctionTermNodes.

IntegerContainerInfo These objects wrap other info objects. Each IntegerContainerTermNode has an IntegerContainerInfo that contains the BasicInfo of the inner TermNode. The name of every IntegerContainerInfo is *INTEGER*.

5.4 DataSet class and TableCls class

Objects of the DataSet class represent predicate interpretations and function interpretations. A DataSet is a list of Tuples, and Tuples are lists of Values. The functions of DataSet are

all for accessing the Tuples and Values or performing simple modifications. No interesting algorithms are implemented by the DataSet class.

Relations are represented by objects of the TableCls class. Each TableCls contains a list of VariableInfo objects and a pointer to a DataSet. For example $P(x, y)$ could be represented by TableCls R_1 and $P(y, x)$ could be represented by TableCls R_2. Both R_1 and R_2 would share the same DataSet, but they would have different variable lists. Unlike BaseFOFormulaNode objects, TableCls objects are designed to be used entirely by algorithms implemented outside the class and its subclasses. No relational algebra operations are implemented as functions of TableCls.

5.5 Input Language

The input for the tool comes in two parts, each with its own language. The *theory file* encodes the formula and includes declarations of the instance predicates and expansion predicates. The *instance file* encodes the instance structure with interpretations for the instance predicates.

The input language used by this version of the tool is identical to the input language used by the original grounder that this version was based upon.

5.5.1 Theory File

The theory file begins with `(` and ends with `)` and everything after the final `)` is ignored. There are three sections: *given*, *find*, and *satisfying*. Each section must be present even if it is empty. The three sections are written `(GIVEN` *givens*`)`, `(FIND` *finds*`)`, and `(SATISFYING` *axioms*`)`, respectively.

The *givens* section contains three subsections and each must be present even if it is empty. The subsection `(TYPES` *types*`)` declares the list of types that will be used, giving each type a name. The subsection `(SYMBOLS` *predicates*`)` declares the names and parameter types of the predicates that are used in the axioms. The subsection `(FUNCTIONS` *functions*`)` declares names, parameter types, and result types for functions. The *givens* section must include all vocabulary; expansion vocabulary must not be excluded.

Each type declaration is written as `(TYPE` *name*`)`, where *name* is a name beginning with either the letter I or the letter S. When the name begins with I it means that this type represents integer values, and S means that the type represents string values. Even though

string values are implemented in the tool, integer values serve the needs of SMT solvers so in practice all types will start with the letter I.

Each predicate declaration is written as `(SYMBOL` *name* `(`*parameters*`))`. The *name* can be any name, and *parameters* is a list of type names without commas. The type names must each have been declared in the types section. This determines how many terms are expected in atoms of the named predicate, and also the size of tuples for the predicate in the instance file.

Each function declaration is written as `(FUNCTION` *name* `(`*parameters*`):` *type*`)`. The *name* must begin with an underscore because that is an enforced naming convention. The *parameters* are a list of type names without commas and *type* is a single type name that represents the type of the result of an instance of the function.

The *finds* section is a list of predicate names without commas. This list represents the expansion vocabulary of the model expansion problem. Function names are not allowed.

The *axioms* section is any number of first-order formulas. These are the formulas to be modeled in the model expansion problem.

5.5.2 First-Order Formula

As with the sections of the theory file, each first-order formula is surrounded by parentheses. Conjunction and disjunction are written `(AND` *children*`)` and `(OR` *children*`)`, where *children* are one or more formulas, each surrounded by its own parentheses to visually separate it from its neighbors. Other formulas that are accepted include `(NOT` ϕ`)`, `(IMPLIES` ϕ_1 ϕ_2`)`, and `(IFF` ϕ_1 ϕ_2`)`, where ϕ, ϕ_1, and ϕ_2 are formulas. The `NOT` form is negation, the `(IMPLIES` ϕ_1 ϕ_2`)` means `(OR (NOT` ϕ_1`)` ϕ_2`)`, and the `(IFF` ϕ_1 ϕ_2`)` means `(AND (IMPLIES` ϕ_1 ϕ_2`) (IMPLIES` ϕ_2 ϕ_1`))`.

Each quantifier is allowed to have any number of variables. You write `(FORALL` $x_1 \ldots x_n$ ϕ`)` or `(EXISTS` $x_1 \ldots x_n$ ϕ`)` where ϕ is a formula and x_1 through x_n are names for the variables to be quantified.

Unlike formulas, terms are not necessarily surrounded by parentheses and terms in a list are separated by commas. An atom is written `(`P`(`t_1`,...,`t_n`))`, where P is the name of a predicate and t_1 through t_n are terms. An atom without terms is written `(`P`())`. A term is written F`(`t_1`,...,`t_n`)`, where F is a function name. A constant is written F`()`.

Arithmetic functions are represented by the function names `+`, `-`, and `*`. These function names do not begin with an underscore because they are built into the tool and not declared

in the theory file, nor interpreted by the instance file. For term t where t is an argument of an arithmetic function, t must be an instance of an arithmetic function, or else t must be an instance of the `INTEGER` function. The `INTEGER` function takes one argument and its result value is exactly the value of its argument. The only purpose of `INTEGER(`x`)` is to allow term x to be included in arithmetic. Internally, an instance of the `INTEGER` function is represented as an IntegerContainerTermNode object.

Comparison predicates are also allowed, such as `(=(`t_1`,`t_2`))`, `(<(`t_1`,`t_2`))`, and `(<=(`t_1`,`t_2`))`. Unlike arithmetic functions, t_1 and t_2 can be any terms. Predicates `>` and `>=` are not allowed.

In addition to being a well formed formula according to the notation of the tool's input language, the grounding procedure only produces correct answers on a fragment of first order logic. All negations must be inside all quantifiers. Otherwise that tool would have to convert an existential quantifier to a universal quantifier or a universal quantifier to an existential quantifier. It does not do that conversion, and the procedure treats universal quantifiers and existential quantifiers very differently.

All existentially quantified variables must have distinct names. During the grounding process new variables are created as necessary and they are all given distinct names by the tool, but an input such as $(\exists y P(x, y)) \wedge (\exists y P(y, x))$ would cause confusion because there are two distinct variables named y. Universally quantified variables have no similar restriction because they are replaced by values during grounding.

5.5.3 Instance File

The instance file provides interpretations for predicates, functions, and types declared in the theory file. There are no sections to this file; it is just an unordered collection of interpretations. Each interpretation is surrounded by parentheses to separate it from the others.

A type interpretation has the form `(TYPE` *name* n `[` n_1 `..` n_2 `])` where *name* is a declared type from the theory file starting with the letter I, n is a numeral indicating the number of values in this type, n_1 and n_2 are numerals indicating the range of integers represented by this type. The numeral n_1 must not be greater than the numeral n_2 and n must equal $n_2 - n_1 + 1$.

A predicate interpretation has the form `(PREDICATE` *name* $p_1 \ldots p_n$`)` where *name* is a declared predicate from the theory file and p_1 through p_n are tuples of numerals matching the parameters listed in the theory file. Tuples are written as `(`r_1`,`$\ldots$`,`r_m`)` where r_1 through

r_m are numerals and m is the arity of the predicate.

A function interpretation has the form `(FUNCTION` *name* $q_1 \ldots q_n$`)` where *name* is a declared function from the theory file and q_1 through q_n are of the form `(`t_1`,...,`t_m`:`r`)`, where t_1 through t_m are numerals representing function parameter values and r is a numeral representing the corresponding term value. For each i with $1 \leq i \leq n$, q_i represents a tuple that can be given to the function an the resulting value of the term, so if the function is a constant that takes no parameters then q_i has the form `(:`r`)`. For example `(FUNCTION _one (:1))` is a way of providing a constant for the number 1 in the axioms of the theory file.

5.6 Second Version

Using what was learned from the construction of the above software tool, we constructed a second version to explore alternate design ideas. While the first version was written in C++ by modifying previously existing software, the second version was written in Java using an entirely new design.

Biggest design change was the removal of almost all responsibilities from the class that represents formulas. Instead of having many subclasses, one for each connective and variety of atom, the Formula class has no subclasses. Each Formula is merely a code to identify the connective, predicate, or quantifier that it represents, a list of variables that is empty for all formulas but quantifiers, and a list of subformulas. Terms are also represented by Formula objects, and it is left to the algorithm to determine which objects are formulas and which are terms from context.

The ability to reduce a partially ground formula involving expansion functions to an SMT formula has also been added. The second version also allows variables with neither quantifiers nor guards, since these variables translate directly to variables in the resulting SMT formula.

The syntax of the allowed input files has been made less restrictive. Underscores are no longer required as a prefix of every function name, and the letter I is no longer required as a prefix for type names.

In order to match the input syntax for a grounding tool that will be used for comparison in a later chapter, quantifiers now require a type for each variable. This sort is internally converted into a guard before grounding, and in general the tool does not distinguish between instance predicates and types.

These design changes were made partially with the goal of improving performance of the SMT solver by simplifying the output formula of the partial grounder. This simplification was difficult to produce with the first version of the tool, but it was successfully achieved in the second version, and so the second version was used in the performance testing for finding magic squares that is discussed later.

Chapter 6

Examples

The following are examples of using partial grounding and an SMT solver to solve model expansion problems. The logic of the SMT solver is assumed to be `QF_LIA`, the quantifier free fragment of first order logic plus the theory of linear integer arithmetic. Addition and comparison vocabulary will be defined by the SMT solver, as will integer numeral constants, but we will also simplify arithmetic whenever possible before giving it to the SMT solver.

6.1 Scheduling

The following is a simple illustration of partial grounding and using SMT to solve a model expansion problem where the expansion vocabulary is a single function of two parameters. Without existential quantifiers in the axioms, the partial grounding is a ground formula as could be produced by any grounding algorithm. This example shows how an expansion function must be converted into a set of variables for the SMT solver to evaluate.

There is a set of jobs $\{J_1, \ldots, J_m\}$ and a list of machines $m_1, \ldots, m_n$. For each job j and machine m, j requires $d(j, m)$ time on m, and if $x < y$ then j must be finished on m_x before it can start on m_y because j must visit each machine in the order that they are listed. Each machine can only process one job at a time.

Given jobs, a list of machines, and function d, we want to find function t such that for each job j and machine m, $t(j, m)$ is the time to start j on m in a schedule that uses the least total time to process all jobs through all machines. To simplify this example, we will merely try to find t such that the total time is less than some constant T and we will use only two machines. The instance predicate J is the set of jobs. The instance constants m_1

and m_2 are the machines, and the predicate M is the set of machines.

$$\sigma = \{d, T, J, M, m_1, m_2\} \quad \varepsilon = \{t\}$$

Axioms:

$$\forall j(J(j) \supset (t(j, M_1) + d(j, M_1) \leq t(j, M_2) \wedge t(j, M_2) + d(j, M_1) \leq T))$$

$$\forall j_1 \forall j_2 \forall m \left((J(j_1) \wedge J(j_2) \wedge M(m) \wedge j_1 < j_2) \supset \vee \begin{array}{l} t(j_1, m) + d(j_1, m) \leq t(j_2, m) \\ t(j_2, m) + d(j_2, m) \leq t(j_1, m) \end{array} \right)$$

An example instance: $m_1 = 1$, $m_2 = 2$, $M = \{1, 2\}$, $J = \{1, 2\}$, $T = 10$,

$$\begin{aligned} d(1,1) &= 1 \\ d(1,2) &= 2 \\ d(2,1) &= 3 \\ d(2,2) &= 4 \end{aligned}$$

This example assumes that expansion functions are allowed. Because of that assumption there are no existential quantifiers and a partial grounding is practically the same a proper grounding.

$$\begin{gathered} t(1,1) + d(1,1) \leq t(1,2) \wedge t(1,2) + d(1,2) \leq 10 \\ t(2,1) + d(2,1) \leq t(2,2) \wedge t(2,2) + d(2,2) \leq 10 \\ t(1,1) + d(1,1) \leq t(2,1) \vee t(2,1) + d(2,1) \leq t(1,1) \\ t(1,2) + d(1,2) \leq t(2,2) \vee t(2,2) + d(2,2) \leq t(1,2) \end{gathered}$$

Reducing the expansion function t to variables and the instance function d to constants, we get the following:

$$\begin{gathered} t_{1,1} + 1 \leq t_{1,2} \wedge t_{1,2} + 2 \leq 10 \\ t_{2,1} + 3 \leq t_{2,2} \wedge t_{2,2} + 4 \leq 10 \\ t_{1,1} + 1 \leq t_{2,1} \vee t_{2,1} + 3 \leq t_{1,1} \\ t_{1,2} + 2 \leq t_{2,2} \vee t_{2,2} + 4 \leq t_{1,2} \end{gathered}$$

Converting t to a set of variables normally requires adding axioms to ensure that the variables obey the rules of a function, but since there are no variables as argument terms for t, the axioms would all simplify to *true*. For example, $(1 = 2 \wedge 1 = 1) \supset t_{1,1} = t_{2,1}$ simplifies as *true*.

6.2 N-Queens

The following is an illustration of partial grounding with an existential quantifier so that the resulting formula is not a ground formula. The variables that exist after partial grounding allows both parts of case 5 in the `Reduce` procedure in Figure 4.1, meaning the positive and the negative instance of an expansion predicate. Without the variable, both positive and negative instances produce the same result. Because of the variable, a literal such as $\neg Q(1,2)$ must be replaced by the negation of a disjunction of variable assignments that could cause $Q(1,2)$ to be true.

The N-Queens problem is putting n queens on an $n \times n$ chess board so that no queen threatens any other by the rules of chess. In other words, no queen is on the same horizontal, vertical, or diagonal line as any other. The expansion predicate $Q(x,y)$ is true if a queen is at position (x,y).

$$\sigma = \{n\} \quad \varepsilon = \{Q\}$$

$$\forall x(1 \leq x \leq n \supset \exists y(1 \leq y \leq n \wedge Q(x,y)))$$

$$\forall x \forall y_1((1 \leq x \leq n \wedge 1 \leq y_1 < n) \supset [Q(x,y_1) \supset \forall y_2(y_1 < y_2 \leq n \supset \neg Q(x,y_2))])$$

$$\forall x_1 \forall y((1 \leq x_1 < n \wedge 1 \leq y \leq n) \supset [Q(x_1,y) \supset \forall x_2(x_1 < x_2 \leq n \supset \neg Q(x_2,y))])$$

$$\forall x_1 \forall x_2 \forall y_1 \forall y_2$$
$$(1 \leq x_1 < x_2 \leq n \wedge 1 \leq y_1 < y_2 \leq n) \supset [(Q(x_1,y_1) \wedge Q(x_2,y_2)) \supset x_2 - x_1 \neq y_2 - y_1]$$

$$\forall x_1 \forall x_2 \forall y_1 \forall y_2$$
$$(1 \leq x_1 < x_2 \leq n \wedge 1 \leq y_2 < y_1 \leq n) \supset [(Q(x_1,y_1) \wedge Q(x_2,y_2)) \supset x_2 - x_1 \neq y_1 - y_2]$$

To prevent the partial grounding from becoming large, let $n = 3$. The partial grounding of the first axiom is:

$$1 \leq y_1 \leq 3 \wedge Q(1,y_1)$$
$$1 \leq y_2 \leq 3 \wedge Q(2,y_2)$$
$$1 \leq y_3 \leq 3 \wedge Q(3,y_3)$$

The partial grounding of the second axiom is:

$$Q(1,1) \supset (\neg Q(1,2) \wedge \neg Q(1,3))$$
$$Q(1,2) \supset \neg Q(1,3)$$
$$Q(2,1) \supset (\neg Q(2,2) \wedge \neg Q(2,3))$$
$$Q(2,2) \supset \neg Q(2,3)$$
$$Q(3,1) \supset (\neg Q(3,2) \wedge \neg Q(3,3))$$
$$Q(3,2) \supset \neg Q(3,3)$$

The partial grounding of the third axiom is:

$$Q(1,1) \supset (\neg Q(2,1) \wedge \neg Q(3,1))$$
$$Q(2,1) \supset \neg Q(3,1)$$
$$Q(1,2) \supset (\neg Q(2,2) \wedge \neg Q(3,2))$$
$$Q(2,2) \supset \neg Q(3,2)$$
$$Q(1,3) \supset (\neg Q(2,3) \wedge \neg Q(3,3))$$
$$Q(2,3) \supset \neg Q(3,3)$$

The partial grounding of the fourth axiom involves comparing every possible pair of coordinates for diagonal threats. For example:

$$(Q(1,1) \wedge Q(2,2)) \supset 2-1 \neq 2-1$$
$$(Q(1,1) \wedge Q(2,3)) \supset 2-1 \neq 3-1$$

The full list of unsimplified grounding lines is too large to write out, but the right side of the $\supset$ of each line can be simplified to $true$ or $false$. Allowing the grounder to perform that step, the fourth axiom becomes:

$$\neg(Q(1,1) \wedge Q(2,2))$$
$$\neg(Q(1,1) \wedge Q(3,3))$$
$$\neg(Q(2,1) \wedge Q(3,2))$$
$$\neg(Q(1,2) \wedge Q(2,3))$$
$$\neg(Q(2,2) \wedge Q(3,3))$$

The grounding of the fifth axiom is similar, and its simplified form is:

$$\neg(Q(1,2) \wedge Q(2,1))$$
$$\neg(Q(2,2) \wedge Q(3,1))$$
$$\neg(Q(1,3) \wedge Q(2,2))$$
$$\neg(Q(1,3) \wedge Q(3,1))$$
$$\neg(Q(2,3) \wedge Q(3,2))$$

To convert to SMT QF_LIA, we replace each predicate atom with a propositional variable. The first axiom becomes:

$$1 \leq y_1 \leq 3 \wedge Q_{1,y_1}$$
$$1 \leq y_2 \leq 3 \wedge Q_{2,y_2}$$
$$1 \leq y_3 \leq 3 \wedge Q_{3,y_3}$$

For the SMT solver, the problem requires values to be supplied for six variables: y_1, y_2, y_3, Q_{1,y_1}, Q_{2,y_2}, Q_{3,y_3}. The above lines ensure that each column of the chess board contains a queen, as represented by Q_{1,y_1}, Q_{2,y_2}, and Q_{3,y_3}. It also ensures that each queen is located between $y = 1$ and $y = 3$, as represented by $1 \leq y_i \leq 3$.

All of the other axioms contain only negative instances of Q, so to convert to SMT all that is needed is to substitute the appropriate subformula for each negative instance. For example $Q(1,1)$ becomes $Q_{1,y_1} \wedge y_1 = 1$ and $Q(2,3)$ becomes $Q_{2,y_2} \wedge y_2 = 3$.

6.3 Edge Matching

6.3.1 Version 1

The following illustrates converting instance vocabulary atoms into disjunctions for the benefit of the SMT solver as described in case 6 of the `Reduce` procedure of Figure 4.1. This case uses instance functions rather than the instance predicates which complicates the procedure by forcing a conversion from something of the form $P(f(x))$ to $\exists y(f(x) = y \wedge P(y))$. Expansion functions becoming variables is illustrated again. The partial grounding results in a ground formula because there are no existential quantifiers and so there are no variable parameters to expansion vocabulary to complicate the `Reduce` procedure.

Given a set of square tile patterns where each side of each tile has a color, fill a $k \times m$ area with instances of those patterns such that adjacent tiles have matching colors on the sides that touch. Tiles can be rotated. Let T be the set of tile patterns and $t(x, s)$ be an instance function such that pattern x has color $t(x, s)$ on side s. Let $p(x, y)$ be an expansion function representing the pattern chosen for position x, y, and $r(x, y)$ be an expansion function representing the rotation chosen for position x, y.

$$\sigma = \{k, m, t, m_4, T\} \quad \varepsilon = \{p, r\}$$

The quantifier-free linear integer arithmetic logic of SMT does not allow modulo, so define the instance function m_4 as follows:

$$\begin{aligned} m_4(0) &= 0 \\ m_4(1) &= 1 \\ m_4(2) &= 2 \\ m_4(3) &= 3 \\ m_4(4) &= 0 \\ m_4(5) &= 1 \\ m_4(6) &= 2 \end{aligned}$$

The function m_4 allows us to wrap the sides of each tile around as it rotates. The domain of $\{0, \ldots, 6\}$ is chosen because sides and rotations vary from 0 to 3, so a side plus a rotation will always be between 0 and 6.

The axioms are as follows:

$$\forall x \forall y((1 \leq x \leq m \wedge 1 \leq y \leq k) \supset (T(p(x,y)) \wedge 0 \leq r(x,y) \leq 3))$$

$$\begin{gathered} \forall x \forall y \\ (1 \leq x \leq m \wedge 1 \leq y < k) \supset \\ t(p(x,y), r(x,y)) = t(p(x,y+1), m_4(2 + r(x,y+1))) \end{gathered}$$

$$\begin{gathered} \forall x \forall y \\ (1 \leq x < m \wedge 1 \leq y \leq k) \supset \\ t(p(x,y), m_4(1 + r(x,y))) = t(p(x+1,y), m_4(3 + r(x+1,y))) \end{gathered}$$

To ground these axioms, choose the following instance for example.

$$m = 2 \quad k = 2 \quad T = \{1, 2\}$$

$$\begin{aligned} t(1,0) &= 0 \qquad t(2,0) = 0 \\ t(1,1) &= 8 \qquad t(2,1) = 0 \\ t(1,2) &= 8 \qquad t(2,2) = 8 \\ t(1,3) &= 8 \qquad t(2,3) = 8 \end{aligned}$$

The first axiom grounds to:

$$\begin{aligned} T(p(1,1)) \wedge 0 \leq r(1,1) \leq 3 \quad & T(p(2,1)) \wedge 0 \leq r(2,1) \leq 3 \\ T(p(1,2)) \wedge 0 \leq r(1,2) \leq 3 \quad & T(p(2,2)) \wedge 0 \leq r(2,2) \leq 3 \end{aligned}$$

With simplification the second axiom grounds to:

$$t(p(1,1), r(1,1)) = t(p(1,2), m_4(2 + r(1,2)))$$
$$t(p(2,1), r(2,1)) = t(p(2,2), m_4(2 + r(2,2)))$$

With simplification the third axiom grounds to:

$$t(p(1,1), m_4(1 + r(1,1))) = t(p(2,1), m_4(3 + r(2,1)))$$
$$t(p(1,2), m_4(1 + r(1,2))) = t(p(2,2), m_4(3 + r(2,2)))$$

Converting the expansion functions into variables creates:

$$T(p_{1,1}) \wedge 0 \leq r_{1,1} \leq 3 \quad T(p_{2,1}) \wedge 0 \leq r_{2,1} \leq 3$$
$$T(p_{1,2}) \wedge 0 \leq r_{1,2} \leq 3 \quad T(p_{2,2}) \wedge 0 \leq r_{2,2} \leq 3$$

$$t(p_{1,1}, r_{1,1}) = t(p_{1,2}, m_4(2 + r_{1,2}))$$
$$t(p_{2,1}, r_{2,1}) = t(p_{2,2}, m_4(2 + r_{2,2}))$$

$$t(p_{1,1}, m_4(1 + r_{1,1})) = t(p_{2,1}, m_4(3 + r_{2,1}))$$
$$t(p_{1,2}, m_4(1 + r_{1,2})) = t(p_{2,2}, m_4(3 + r_{2,2}))$$

To finish the conversion to an SMT axiomatization, we need to simplify the instance predicates and functions. For the first four formulas, this is simply:

$$(p_{1,1} = 1 \vee p_{1,1} = 2) \wedge 0 \leq r_{1,1} \leq 3 \quad (p_{2,1} = 1 \vee p_{2,1} = 2) \wedge 0 \leq r_{1,1} \leq 3$$
$$(p_{1,2} = 1 \vee p_{1,2} = 2) \wedge 0 \leq r_{1,1} \leq 3 \quad (p_{2,2} = 1 \vee p_{2,2} = 2) \wedge 0 \leq r_{1,1} \leq 3$$

For the remaining formulas, we first replace the instance functions with equivalent atoms because the procedure does not deal with terms. It is simple to substitute a variable for each term and create a conjunction that equates the variable with the term as follows:

$$t(p_{1,1}, r_{1,1}) = t_1 \wedge m_4(2 + r_{1,2}) = x_1 \wedge t(p_{1,2}, x_1) = t_2 \wedge t_1 = t_2$$
$$t(p_{2,1}, r_{2,1}) = t_3 \wedge m_4(2 + r_{2,2}) = x_2 \wedge t(p_{2,2}, x_2) = t_4 \wedge t_3 = t_4$$

$$m_4(1 + r_{1,1}) = x_3 \wedge t(p_{1,1}, x_3) = t_5 \wedge m_4(3 + r_{2,1}) = x_4 \wedge t(p_{2,1}, x_4) = t_6 \wedge t_5 = t_6$$
$$m_4(1 + r_{1,2}) = x_5 \wedge t(p_{1,2}, x_5) = t_7 \wedge m_4(3 + r_{2,2}) = x_6 \wedge t(p_{2,2}, x_6) = t_8 \wedge t_7 = t_8$$

Each of the instance function atoms is then substituted with an equivalent disjunction based on its definition in the instance structure. The resulting formula is too large to include

in full, but for example $t(p_{1,1}, r_{1,1}) = t_1$ is substituted with

$$\bigvee \left\{ \begin{array}{l} p_{1,1} = 1 \wedge r_{1,1} = 0 \wedge t_1 = 0 \\ p_{1,1} = 1 \wedge r_{1,1} = 1 \wedge t_1 = 8 \\ p_{1,1} = 1 \wedge r_{1,1} = 2 \wedge t_1 = 8 \\ p_{1,1} = 1 \wedge r_{1,1} = 3 \wedge t_1 = 8 \\ p_{1,1} = 2 \wedge r_{1,1} = 0 \wedge t_1 = 0 \\ p_{1,1} = 2 \wedge r_{1,1} = 1 \wedge t_1 = 0 \\ p_{1,1} = 2 \wedge r_{1,1} = 2 \wedge t_1 = 8 \\ p_{1,1} = 2 \wedge r_{1,1} = 3 \wedge t_1 = 8 \end{array} \right\}$$

The other substitutions for t are similar and vary only by the names of the variables. For m_4, take $m_4(2 + r_{1,2}) = x_1$ as an example:

$$\bigvee \left\{ \begin{array}{l} 2 + r_{1,2} = 0 \wedge x_1 = 0 \\ 2 + r_{1,2} = 1 \wedge x_1 = 1 \\ 2 + r_{1,2} = 2 \wedge x_1 = 2 \\ 2 + r_{1,2} = 3 \wedge x_1 = 3 \\ 2 + r_{1,2} = 4 \wedge x_1 = 0 \\ 2 + r_{1,2} = 5 \wedge x_1 = 1 \\ 2 + r_{1,2} = 6 \wedge x_1 = 2 \end{array} \right\}$$

6.3.2 Version 2

The following is an example where the SMT solver is required to solve linear integer arithmetic of the form $x + 1 = y$ which is an instance of summation that cannot be simplified away. The variables involved are generated to represent expansion functions. There are no existential quantifiers to produce variables after partial grounding. Converting instance vocabulary to disjunctions is also illustrated.

Given a set of square tiles where each side of each tile has a color, lay the tiles within a $k \times m$ area such that adjacent tiles have matching colors on the sides that touch. Tiles can be rotated. Let T be the set of tiles and $t(w, s)$ be an instance function such that tile w has color $t(w, s)$ on side s. Let $x(w)$ and $y(w)$ be expansion functions representing that tile w is laid at coordinates $(x(w), y(w))$, and $r(w)$ be an expansion function representing the rotation chosen for tile w.

$$\sigma = \{k, m, t, m_4, T\} \quad \varepsilon = \{x, y, r\}$$

The quantifier-free linear integer arithmetic logic of SMT does not allow modulo, so define the instance function m_4 as follows:

$$m_4(0) = 0$$
$$m_4(1) = 1$$
$$m_4(2) = 2$$
$$m_4(3) = 3$$
$$m_4(4) = 0$$
$$m_4(5) = 1$$
$$m_4(6) = 2$$

The function m_4 allows us to wrap the sides of each tile around as it rotates. The domain of $\{0, \ldots, 6\}$ is chosen because sides and rotations vary from 0 to 3, so a side plus a rotation will always be between 0 and 6.

The axioms are as follows:

$$\forall w(T(w) \supset (1 \leq x(w) \leq m \wedge 1 \leq y(w) \leq k \wedge 0 \leq r(w) \leq 3))$$

$$\forall w_1 \forall w_2$$
$$(T(w_1) \wedge T(w_2) \wedge w_1 \neq w_2) \supset$$
$$(x(w_1) = x(w_2) \wedge y(w_1) + 1 = y(w_2)) \supset t(w_1, r(w_1)) = t(w_2, m_4(2 + r(w_2)))$$

$$\forall w_1 \forall w_2$$
$$(T(w_1) \wedge T(w_2) \wedge w_1 \neq w_2) \supset$$
$$(x(w_1) + 1 = x(w_2) \wedge y(w_1) = y(w_2)) \supset t(w_1, m_4(1 + r(w_1))) = t(w_2, m_4(3 + r(w_2)))$$

To ground these axioms, choose the following instance for example.

$$m = 2 \quad k = 2 \quad T = \{1, 2, 3\}$$

$$t(1,0) = 0 \qquad t(2,0) = 0 \qquad t(2,0) = 7$$
$$t(1,1) = 8 \qquad t(2,1) = 0 \qquad t(2,1) = 8$$
$$t(1,2) = 8 \qquad t(2,2) = 8 \qquad t(2,2) = 8$$
$$t(1,3) = 8 \qquad t(2,3) = 8 \qquad t(2,3) = 7$$

The simplifying the constants, grounding for the first axiom is:

$$1 \leq x(1) \leq 2 \wedge 1 \leq y(1) \leq 2 \wedge 0 \leq r(1) \leq 3$$
$$1 \leq x(2) \leq 2 \wedge 1 \leq y(2) \leq 2 \wedge 0 \leq r(2) \leq 3$$
$$1 \leq x(3) \leq 2 \wedge 1 \leq y(3) \leq 2 \wedge 0 \leq r(3) \leq 3$$

The grounding for the second axiom is:

$$(x(1) = x(2) \wedge y(1) + 1 = y(2)) \supset t(1, r(1)) = t(2, m_4(2 + r(2)))$$
$$(x(1) = x(3) \wedge y(1) + 1 = y(3)) \supset t(1, r(1)) = t(3, m_4(2 + r(3)))$$
$$(x(2) = x(1) \wedge y(2) + 1 = y(1)) \supset t(2, r(2)) = t(1, m_4(2 + r(1)))$$
$$(x(2) = x(3) \wedge y(2) + 1 = y(3)) \supset t(2, r(2)) = t(3, m_4(2 + r(3)))$$
$$(x(3) = x(1) \wedge y(3) + 1 = y(1)) \supset t(3, r(3)) = t(1, m_4(2 + r(1)))$$
$$(x(3) = x(2) \wedge y(3) + 1 = y(2)) \supset t(3, r(3)) = t(2, m_4(2 + r(2)))$$

The grounding for the third axiom is:

$$(x(1) + 1 = x(2) \wedge y(1) = y(2)) \supset t(1, m_4(1 + r(1))) = t(2, m_4(3 + r(2)))$$
$$(x(1) + 1 = x(3) \wedge y(1) = y(3)) \supset t(1, m_4(1 + r(1))) = t(3, m_4(3 + r(3)))$$
$$(x(2) + 1 = x(1) \wedge y(2) = y(1)) \supset t(2, m_4(1 + r(2))) = t(1, m_4(3 + r(1)))$$
$$(x(2) + 1 = x(3) \wedge y(2) = y(3)) \supset t(2, m_4(1 + r(2))) = t(3, m_4(3 + r(3)))$$
$$(x(3) + 1 = x(1) \wedge y(3) = y(1)) \supset t(3, m_4(1 + r(3))) = t(1, m_4(3 + r(1)))$$
$$(x(3) + 1 = x(2) \wedge y(3) = y(2)) \supset t(3, m_4(1 + r(3))) = t(2, m_4(3 + r(2)))$$

By converting all the expansion functions to variables we get:

$$1 \leq x_1 \leq 2 \wedge 1 \leq y_1 \leq 2 \wedge 0 \leq r_1 \leq 3$$
$$1 \leq x_2 \leq 2 \wedge 1 \leq y_2 \leq 2 \wedge 0 \leq r_2 \leq 3$$
$$1 \leq x_3 \leq 2 \wedge 1 \leq y_3 \leq 2 \wedge 0 \leq r_3 \leq 3$$

$$(x_1 = x_2 \wedge y_1 + 1 = y_2) \supset t(1, r_1) = t(2, m_4(2 + r_2))$$
$$(x_1 = x_3 \wedge y_1 + 1 = y_3) \supset t(1, r_1) = t(3, m_4(2 + r_3))$$
$$(x_2 = x_1 \wedge y_2 + 1 = y_1) \supset t(2, r_2) = t(1, m_4(2 + r_1))$$
$$(x_2 = x_3 \wedge y_2 + 1 = y_3) \supset t(2, r_2) = t(3, m_4(2 + r_3))$$
$$(x_3 = x_1 \wedge y_3 + 1 = y_1) \supset t(3, r_3) = t(1, m_4(2 + r_1))$$
$$(x_3 = x_2 \wedge y_3 + 1 = y_2) \supset t(3, r_3) = t(2, m_4(2 + r_2))$$

$$(x_1 + 1 = x_2 \wedge y_1 = y_2) \supset t(1, m_4(1 + r_1)) = t(2, m_4(3 + r_2))$$
$$(x_1 + 1 = x_3 \wedge y_1 = y_3) \supset t(1, m_4(1 + r_1)) = t(3, m_4(3 + r_3))$$
$$(x_2 + 1 = x_1 \wedge y_2 = y_1) \supset t(2, m_4(1 + r_2)) = t(1, m_4(3 + r_1))$$
$$(x_2 + 1 = x_3 \wedge y_2 = y_3) \supset t(2, m_4(1 + r_2)) = t(3, m_4(3 + r_3))$$
$$(x_3 + 1 = x_1 \wedge y_3 = y_1) \supset t(3, m_4(1 + r_3)) = t(1, m_4(3 + r_1))$$
$$(x_3 + 1 = x_2 \wedge y_3 = y_2) \supset t(3, m_4(1 + r_3)) = t(2, m_4(3 + r_2))$$

We must replace the instance functions with equivalent atoms because the procedure does not deal with terms. It is simple to substitute a variable for each term and create a conjunction that equates the variable with the term as follows:

$$(x_1 + 1 = x_2 \wedge y_1 = y_2) \supset t(1, m_4(1 + r_1)) = t(2, m_4(3 + r_2))$$
becomes
$$(x_1 + 1 = x_2 \wedge y_1 = y_2) \supset$$
$$m_4(1 + r_1) = a_1 \wedge t(1, a_1) = t_1 \wedge m_4(3 + r_2) = a_2 \wedge t(2, a_2) = t_2 \wedge t_1 = t_2$$

The entire transformation is too large to include in full, but each line is transformed similarly. Each of the instance function atoms is then substituted with an equivalent disjunction based on its definition in the instance structure. Continuing with the above example, $m_4(1+r_1) = a_1$ is substituted with:

$$\bigvee \left\{ \begin{array}{l} 2 + r_{1,2} = 0 \wedge x_1 = 0 \\ 2 + r_{1,2} = 1 \wedge x_1 = 1 \\ 2 + r_{1,2} = 2 \wedge x_1 = 2 \\ 2 + r_{1,2} = 3 \wedge x_1 = 3 \\ 2 + r_{1,2} = 4 \wedge x_1 = 0 \\ 2 + r_{1,2} = 5 \wedge x_1 = 1 \\ 2 + r_{1,2} = 6 \wedge x_1 = 2 \end{array} \right\}$$

The including simplification, atom $t(1, a_1) = t_1$ is substituted with:

$$\bigvee \left\{ \begin{array}{l} a_1 = 1 \wedge t_1 = 0 \\ a_1 = 2 \wedge t_1 = 8 \\ a_1 = 3 \wedge t_1 = 8 \\ a_1 = 4 \wedge t_1 = 8 \end{array} \right\}$$

The other substitutions are similar, varying only by the names of the variables.

6.4 Hydraulic Planning

The following example expands upon the use of case 5 of the `Reduce` procedure of Figure 4.1 already illustrated using N-Queens in Section 6.2. It contains several existential quantifiers and multiple expansion predicates that appear in both positive and negative instances with variable arguments. Case 6 disjunction substitutions are never needed for the instance predicates because they are used always either as guards or with constant arguments after partial grounding. It makes no use of arithmetic.

A directed graph represents connected pipes, valves, and tanks. Each edge represents a valve that can be either open or closed. Each node in V represents a pipe or other space where fluid may reside. If a node x is a pressurized tank then instance predicate $T(x)$ will be true. If valve x, y is initially closed then instance predicate $C(x, y)$ will be true and if it is initially open then $O(x, y)$ will be true. If a node is an empty tank that needs to be filled, then instance predicate $G(x)$ will be true. Find a set of no more than n valves to open so that all tanks in G are filled from some tank in T.

The expansion predicate $P(x, y, i)$ means that node y is pressurized from node x after step i. The expansion predicate $A(x, y)$ is the set of valves that are to be opened.

$$\sigma = \{V, T, C, O, G\} \quad \varepsilon = \{P, A\}$$

The axioms are as follows:

$$\forall x \forall y \forall i((V(x) \wedge V(y) \wedge 1 \leq i \leq n) \supset ((\neg O(x, y) \wedge \neg C(x, y)) \supset \neg P(x, y, i)))$$

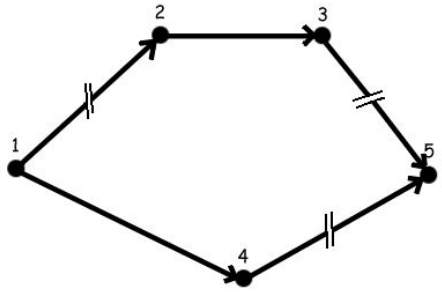
$$\forall x_0 \forall y_0 \forall x_1 \forall y_1 \forall i$$
$$(C(x_0, y_0) \wedge C(x_1, y_1) \wedge 1 \leq i \leq n) \supset ((P(x_0, y_0, i) \wedge P(x_1, y_1, i)) \supset (x_0 = x_1 \wedge y_0 = y_1))$$

$$\forall x \forall y \forall i$$
$$(O(x, y) \wedge 1 \leq i \leq n) \supset (P(x, y, i) \supset (T(x) \vee \exists j \exists w(V(w) \wedge 1 \leq j \leq i \wedge P(w, x, j))))$$

$$\forall x \forall y \forall i$$
$$(C(x, y) \wedge 1 \leq i \leq n) \supset (P(x, y, i) \supset (T(x) \vee \exists j \exists w(V(w) \wedge 1 \leq j < i \wedge P(w, x, j))))$$
$$\forall y(G(y) \supset \exists x \exists i(V(x) \wedge 1 \leq i \leq n \wedge P(x, y, i)))$$

$$\forall x \forall y(C(x, y) \supset (A(x, y) \leftrightarrow \exists i(0 \leq i \leq n \wedge P(x, y, i))))$$

For this example, the instance structure will be defined as follows:



$$V = \{1, 2, 3, 4, 5\}$$
$$O = \{(2, 3), (1, 4)\}$$
$$C = \{(1, 2), (3, 5), (4, 5)\}$$
$$T = \{1\}$$
$$G = \{5\}$$
$$n = 2$$

After simplifying the instance predicates O and C, the grounding of the first axiom becomes:

$$\begin{array}{llll}
\neg P(1,1,1) & \neg P(1,1,2) & \neg P(2,2,1) & \neg P(2,2,2) \\
\neg P(3,3,1) & \neg P(3,3,2) & \neg P(4,4,1) & \neg P(4,4,2) \\
\neg P(5,5,1) & \neg P(5,5,2) & & \\
\neg P(2,1,1) & \neg P(2,1,2) & \neg P(3,1,1) & \neg P(3,1,2) \\
\neg P(4,1,1) & \neg P(4,1,2) & \neg P(5,1,1) & \neg P(5,1,2) \\
\neg P(3,2,1) & \neg P(3,2,2) & \neg P(4,2,1) & \neg P(4,2,2) \\
\neg P(5,2,1) & \neg P(5,2,2) & & \\
\neg P(4,3,1) & \neg P(4,3,2) & \neg P(5,3,1) & \neg P(5,3,2) \\
\neg P(5,4,1) & \neg P(5,4,2) & & \\
\neg P(1,3,1) & \neg P(1,3,2) & \neg P(1,5,1) & \neg P(1,5,2) \\
\neg P(2,4,1) & \neg P(2,4,2) & \neg P(3,4,1) & \neg P(3,4,2)
\end{array}$$

The simplifying away cases where $x_0 = x_1$ and $y_0 = y_1$ and cases which differ only by conjunct order, the grounding for the second axiom becomes:

$$\begin{array}{ll}
\neg(P(1,2,1) \wedge P(3,5,1)) & \neg(P(1,2,2) \wedge P(3,5,2)) \\
\neg(P(1,2,1) \wedge P(4,5,1)) & \neg(P(1,2,2) \wedge P(4,5,2)) \\
\neg(P(3,5,1) \wedge P(4,5,1)) & \neg(P(3,5,2) \wedge P(4,5,2))
\end{array}$$

The grounding of the third axiom is as follows:

$$
\begin{aligned}
P(2,3,1) &\supset (T(2) \vee (V(w_1) \wedge 1 \leq j_1 \leq 1 \wedge P(w_1,2,j_1))) \\
P(2,3,2) &\supset (T(2) \vee (V(w_2) \wedge 1 \leq j_2 \leq 2 \wedge P(w_2,2,j_2))) \\
P(1,4,1) &\supset (T(1) \vee (V(w_3) \wedge 1 \leq j_3 \leq 1 \wedge P(w_3,2,j_3))) \\
P(1,4,2) &\supset (T(1) \vee (V(w_4) \wedge 1 \leq j_4 \leq 2 \wedge P(w_4,2,j_4)))
\end{aligned}
$$

Simplifying the above grounding with the definition of T gives:

$$
\begin{aligned}
P(2,3,1) &\supset (V(w_1) \wedge 1 \leq j_1 \leq 1 \wedge P(w_1,2,j_1)) \\
P(2,3,2) &\supset (V(w_2) \wedge 1 \leq j_2 \leq 2 \wedge P(w_2,2,j_2))
\end{aligned}
$$

The atoms $P(w_1,2,j_1)$ and $P(w_2,2,j_2)$ are positive instance of expansion predicate P. All positive instance of expansion predicates must be noted for use when converting the grounding into an SMT problem.

The grounding of the forth axiom is as follows:

$$
\begin{aligned}
P(1,2,1) &\supset (T(1) \vee (V(w_3) \wedge 1 \leq j_3 < 1 \wedge P(w_3,1,j_3))) \\
P(1,2,2) &\supset (T(1) \vee (V(w_4) \wedge 1 \leq j_4 < 2 \wedge P(w_4,1,j_4))) \\
P(3,5,1) &\supset (T(3) \vee (V(w_5) \wedge 1 \leq j_5 < 1 \wedge P(w_5,3,j_5))) \\
P(3,5,2) &\supset (T(3) \vee (V(w_6) \wedge 1 \leq j_6 < 2 \wedge P(w_6,3,j_6))) \\
P(4,5,1) &\supset (T(4) \vee (V(w_7) \wedge 1 \leq j_7 < 1 \wedge P(w_7,4,j_7))) \\
P(4,5,2) &\supset (T(4) \vee (V(w_8) \wedge 1 \leq j_8 < 2 \wedge P(w_8,4,j_8)))
\end{aligned}
$$

With simplification, the above grounding becomes:

$$
\begin{aligned}
&\neg P(3,5,1) \\
P(3,5,2) &\supset (V(w_6) \wedge 1 \leq j_6 < 2 \wedge P(w_6,3,j_6)) \\
&\neg P(4,5,1) \\
P(4,5,2) &\supset (V(w_8) \wedge 1 \leq j_8 < 2 \wedge P(w_8,4,j_8))
\end{aligned}
$$

The atoms $P(w_6,3,j_6)$ and $P(w_8,4,j_8)$ are the positive expansion predicate instances in the above grounding.

The grounding of the fifth axiom is as follows:

$$V(x) \wedge 1 \leq i \leq 2 \wedge P(x,5,i)$$

The atom $P(x,5,i)$ must be noted as a positive instance.

The grounding of the sixth axiom is as follows:

$$A(1,2) \leftrightarrow (0 \leq i_1 \leq 5 \wedge P(1,2,i_1))$$
$$A(3,5) \leftrightarrow (0 \leq i_2 \leq 5 \wedge P(3,5,i_2))$$
$$A(4,5) \leftrightarrow (0 \leq i_3 \leq 5 \wedge P(4,5,i_3))$$

The atoms $A(1,2)$, $A(3,5)$, $A(4,5)$, $P(1,2,i_1)$, $P(3,5,i_2)$, and $P(4,5,i_3)$ must be noted as positive instances of expansion predicates. Because of the $\leftrightarrow$ connective, all of the atoms are both positive and negative as is clear when the connective is expanded into a conjunction of disjunctions. This expansion must be performed because the negative instances will need a different substitution than the positive instances.

All positive instance of expansion predicates must be substituted with propositional variables which represent the truth of the original atom. So that the meanings of the new variables are clear, they are named according to the original atom. For example, $P(w_1,2,j_1)$ will become $P_{w_1,2,j_1}$ which is a propositional variable whose truth no longer depends upon w_1 or j_1 because those variable names are the subscript; they are not standing in for some other subscript. Similarly, $A(1,2)$ will become $A_{1,2}$.

Each negative instance must be substituted with a subformula in terms of the positive instances of that expansion predicate. For example, $P(3,2,1)$ appears negatively in the grounding of the first axioms. The following subformula would be substituted for that atom:

$$\bigvee \left\{ \begin{array}{c} P_{w_1,2,j_1} \wedge w_1 = 3 \wedge 2 = 2 \wedge j_i = 1 \\ P_{w_2,2,j_2} \wedge w_2 = 3 \wedge 2 = 2 \wedge j_2 = 1 \\ P_{w_6,3,j_6} \wedge w_6 = 3 \wedge 3 = 2 \wedge j_6 = 1 \\ P_{w_8,4,j_8} \wedge w_8 = 3 \wedge 4 = 2 \wedge j_8 = 1 \\ P_{1,2,i_1} \wedge 1 = 3 \wedge 2 = 2 \wedge i_1 = 1 \\ P_{3,5,i_2} \wedge 3 = 3 \wedge 5 = 2 \wedge i_2 = 1 \\ P_{4,5,i_3} \wedge 4 = 3 \wedge 5 = 2 \wedge i_3 = 1 \end{array} \right\}$$

Several simplifications should be made to the above formula before it is given to the SMT solver, but as it is it serves to illustrate how every positive instance of P must be considered when substituting for each negative instance of P. In the case of A, the negative instances become the same as the positive instances after simplification.

6.5 Magic Square

The example of a magic square is intended to illustrate summation in a model expansion problem as the rows, columns, and diagonals are summed. Partial grounding produces a ground formula because there are no existential quantifiers, but expansion functions cause variables in the SMT formula and a large portion of the SMT formula is occupied with arithmetic.

A magic square is an $n \times n$ matrix of integers from 1 to n^2 such that all columns and all rows sum to the same integer. The two diagonals must also sum to that integer. To axiomatize the finding of a magic square of size n for model expansion, the instance structure consists only of the constant n. The expansion structure has the function $f(x, y)$ which represents the magic square and the constant c which represents the sum of any row or column. For bookkeeping purposes, we will also use $h(x, y)$, $j(x, y)$, $k_1(y)$, and $k_2(y)$. The function $h(x, y)$ represents the sum along row y from column 1 to column x. The function $j(x, y)$ represents the sum along column x from row 1 to row y. The function $k_1(y)$ represents the sum along the diagonal from $(1, 1)$ to row y. The function $k_2(y)$ represents the sum along the diagonal from $(n, 1)$ to row y.

$$\sigma = \{n\} \quad \varepsilon = \{c, f, h, j, k_1, k_2\}$$

The axiomatization for finding a magic square is as follows:

$$\forall x \forall y((1 \leq x \leq n \wedge 1 \leq y \leq n) \supset 1 \leq f(x, y) \leq n^2)$$

$$\begin{gathered}\forall x_1 \forall y_1 \forall x_2 \forall y_2 \\ (1 \leq x_1 \leq n \wedge 1 \leq y \leq n \wedge 1 \leq x_2 \leq n \wedge 1 \leq y_2 n) \supset \\ x_1 \neq x_2 \wedge y_1 \neq y_2 \supset f(x_1, y_1) \neq f(x_2, y_2)\end{gathered}$$

$$\begin{gathered}\forall y(1 \leq y \leq n \supset h(1, y) = f(1, y)) \\ \forall x \forall y((2 \leq x \leq n \wedge 1 \leq y \leq n) \supset h(x, y) = h(x-1, y) + f(x, y)) \\ \forall y(1 \leq y \leq n \supset h(n, y) = c)\end{gathered}$$

$$\begin{gathered}\forall x(1 \leq x \leq n \supset j(x, 1) = f(x, 1)) \\ \forall x \forall y((1 \leq x \leq n \wedge 2 \leq y \leq n) \supset j(x, y) = j(x, y-1) + f(x, y)) \\ \forall x(1 \leq x \leq n \supset j(x, n) = c)\end{gathered}$$

$$k_1(1) = f(1,1) \wedge k_2(1) = f(n,1)$$
$$\forall i(2 \leq i \leq n \supset k_1(i) = k_1(i-1) + f(i,i))$$
$$\forall i(2 \leq i \leq n \supset k_2(i) = k_2(i-1) + f(n-i+1,i))$$
$$k_1(n) = c \wedge k_2(n) = c$$

To ground this example, we will use the instance $n = 3$. The first axiom grounds to 9 similar instances of $1 \leq f(x,y) \leq 9$ for every x and y between 1 and 3, ensuring that no illegal numbers will appear in the square. With simplification, the second axiom grounds to 36 similar instances of $f(x_1,y_1) \neq f(x_2,y_2)$ for every relevant x_1, y_1, x_2, y_2 to ensure that no number is used more than once. The remaining axioms are organized into three groups, one for each of h, j, and k.

The grounding for the h axioms is as follows:

$$h(1,1) = f(1,1)$$
$$h(1,2) = f(1,2)$$
$$h(1,3) = f(1,3)$$
$$h(2,1) = h(1,1) + f(2,1) \quad h(3,1) = h(2,1) + f(3,1)$$
$$h(2,2) = h(1,2) + f(2,2) \quad h(3,2) = h(2,2) + f(3,2)$$
$$h(2,3) = h(1,3) + f(2,3) \quad h(3,3) = h(2,3) + f(3,3)$$
$$h(3,1) = c$$
$$h(3,2) = c$$
$$h(3,3) = c$$

The grounding for the j axioms is as follows:

$$j(1,1) = f(1,1)$$
$$j(2,1) = f(2,1)$$
$$j(3,1) = f(3,1)$$
$$j(1,2) = j(1,1) + f(1,2) \quad j(1,3) = j(1,2) + f(1,3)$$
$$j(2,2) = j(2,1) + f(2,2) \quad j(2,3) = j(2,2) + f(2,3)$$
$$j(3,2) = j(3,1) + f(3,2) \quad j(3,3) = j(3,2) + f(3,3)$$
$$j(1,3) = c$$
$$j(2,3) = c$$
$$j(3,3) = c$$

The grounding for the k axioms is as follows:

$$k_1(1) = f(1,1) \wedge k_2(1) = f(3,1)$$
$$k_1(2) = k_1(1) + f(2,2) \quad k_1(3) = k_1(2) + f(3,3)$$
$$k_2(2) = k_2(1) + f(2,2) \quad k_2(3) = k_2(2) + f(1,3)$$
$$k_1(3) = c \wedge k_2(3) = c$$

The conversion of the above grounding to an SMT formula is a trivial matter of replacing each expansion function term with an SMT variable. For example, x could be substituted for all occurrences of $f(1,1)$ and y could be substituted for all occurrences of $f(1,2)$. We often write the variable name for $f(1,1)$ as $f_{1,1}$ to make it easy to recognize, but subscripts are not actually accepted in the SMT-LIB Standard, so some variable names must be chosen for all 34 variables of the grounding. Once the variable names have been chosen, the grounding is ready for the SMT solver.

Chapter 7

Demonstration

In this chapter partial grounding is demonstrated using an SMT solver called Yices[26] and the partial grounder of Chapter 5. The first section uses the n-queens problem and the second section uses the hydraulic planning problem from Section 6.4, both using the first version of the partial grounder. In the third section we use the magic square problem to compare between a Yices solution running time using partial grounding and a SAT-solver solution running time. The second version of the partial grounder is used for best performance.

The output of the version information for Yices is as follows.

```
Yices 2.0 prototype. Copyright SRI International, 2009
GMP 4.2.1. Copyright Free Software Foundation, Inc.
Build date: Thu Jul 29 14:32:44 PDT 2010
Platform: i686-pc-mingw32 (release)
```

7.1 N-Queens

The n-queens problem is given to the grounder in the formula file. Using the notation that the grounder requires, the formula file is as follows.

```
(
  (GIVEN
    (TYPES (TYPE INum))
    (SYMBOLS
      (SYMBOL Queen(INum INum))
```

```
  )
  (FUNCTIONS )
)
(FIND Queen)
(SATISFYING
  (FORALL x (EXISTS y (Queen(x,y))))
  (FORALL x1 y1 x2 y2
    (OR
      (NOT
        (AND
          (< (x2, x1))
          (Queen(x1, y1))
          (Queen(x2, y2))
        )
      )
      (NOT
        (= (-(INTEGER(x2),INTEGER(x1)), -(INTEGER(y2),INTEGER(y1))))
      )
    )
  )
  (FORALL x1 y1 x2 y2
    (OR
      (NOT
        (AND
          (< (y2, y1))
          (Queen(x1, y1))
          (Queen(x2, y2))
        )
      )
      (NOT
        (= (-(INTEGER(x1),INTEGER(x2)), -(INTEGER(y2),INTEGER(y1))))
      )
    )
  )
  (FORALL x y1
    (OR
      (NOT (Queen(x,y1)))
```

```
        (FORALL y2 (OR (NOT (Queen(x,y2))) (=(y2,y1))))
      )
    )
    (FORALL x1 y
      (OR
        (NOT (Queen(x1,y)))
        (FORALL x2 (OR (NOT (Queen(x2,y))) (=(x2,x1))))
      )
    )
  )
)
```

The following instance file says that we want a solution for 4-Queens. It defines the integers to be $1, 2, 3, 4$.

```
(TYPE INum 4[1..4])
```

The result of the grounder is a long SMT formula in the standard notation that Yices accepts. It is too long to include here in full, but here are several excerpts.

```
(and
(and (<= 1 Y.1) (<= Y.1 4))
(and (<= 1 Y.2) (<= Y.2 4))
(and (<= 1 Y.3) (<= Y.3 4))
(and (<= 1 Y.4) (<= Y.4 4))
)
```

The grounder knows that the variables involved are of type INum, so it starts by ensuring that the SMT solver knows it as well. Without this, the SMT solver might assign the variables any integer. The variables `Y.1` through `Y.4` were generated by the grounder because of `(FORALL x (EXISTS y (Queen(x,y))))`, which grounds to four versions of `(Queen(`i`,Y.`i`))`, each with a different i. This is given to the SMT solver as the following.

```
(and Queen_1_Y.1 Queen_2_Y.2 Queen_3_Y.3 Queen_4_Y.4)
```

All four columns must have a queen, so all four propositions must be true. The interesting part of the model produced by the SMT solver will be the values of `Y.1` through `Y.4`, which

indicates the position of each queen. The remainder of the SMT formulas are devoted to preventing illegal arrangements of queens. For example, the following two conjuncts prevent $\texttt{Queen}(2,2) \wedge \texttt{Queen}(1,1)$ and $\texttt{Queen}(3,3) \wedge \texttt{Queen}(1,1)$. I have added the indentation by hand.

```
(not
  (and
    (or
      (and Queen_1_Y.1 (= 2 1) (= 2 Y.1))
      (and Queen_2_Y.2 (= 2 2) (= 2 Y.2))
      (and Queen_3_Y.3 (= 2 3) (= 2 Y.3))
      (and Queen_4_Y.4 (= 2 4) (= 2 Y.4)))
    (or
      (and Queen_1_Y.1 (= 1 1) (= 1 Y.1))
      (and Queen_2_Y.2 (= 1 2) (= 1 Y.2))
      (and Queen_3_Y.3 (= 1 3) (= 1 Y.3))
      (and Queen_4_Y.4 (= 1 4) (= 1 Y.4)))))
(not
  (and
    (or
      (and Queen_1_Y.1 (= 3 1) (= 3 Y.1))
      (and Queen_2_Y.2 (= 3 2) (= 3 Y.2))
      (and Queen_3_Y.3 (= 3 3) (= 3 Y.3))
      (and Queen_4_Y.4 (= 3 4) (= 3 Y.4)))
    (or
      (and Queen_1_Y.1 (= 1 1) (= 1 Y.1))
      (and Queen_2_Y.2 (= 1 2) (= 1 Y.2))
      (and Queen_3_Y.3 (= 1 3) (= 1 Y.3))
      (and Queen_4_Y.4 (= 1 4) (= 1 Y.4)))))
```

The grounding for the formulas that prevent two queens on the same row or column is more elaborate because of the way the original formula was designed. Here are the first two conjuncts, with indentation added by hand.

```
(or
  (not (or
    (and Queen_1_Y.1 (= 1 1) (= 1 Y.1))
    (and Queen_2_Y.2 (= 1 2) (= 1 Y.2))
    (and Queen_3_Y.3 (= 1 3) (= 1 Y.3))
```

```
    (and Queen_4_Y.4 (= 1 4) (= 1 Y.4))))
  (and
    (not (or
      (and Queen_1_Y.1 (= 1 1) (= 2 Y.1))
      (and Queen_2_Y.2 (= 1 2) (= 2 Y.2))
      (and Queen_3_Y.3 (= 1 3) (= 2 Y.3))
      (and Queen_4_Y.4 (= 1 4) (= 2 Y.4))))
    (not (or
      (and Queen_1_Y.1 (= 1 1) (= 3 Y.1))
      (and Queen_2_Y.2 (= 1 2) (= 3 Y.2))
      (and Queen_3_Y.3 (= 1 3) (= 3 Y.3))
      (and Queen_4_Y.4 (= 1 4) (= 3 Y.4))))
    (not (or
      (and Queen_1_Y.1 (= 1 1) (= 4 Y.1))
      (and Queen_2_Y.2 (= 1 2) (= 4 Y.2))
      (and Queen_3_Y.3 (= 1 3) (= 4 Y.3))
      (and Queen_4_Y.4 (= 1 4) (= 4 Y.4))))))
(or
  (not (or
    (and Queen_1_Y.1 (= 1 1) (= 2 Y.1))
    (and Queen_2_Y.2 (= 1 2) (= 2 Y.2))
    (and Queen_3_Y.3 (= 1 3) (= 2 Y.3))
    (and Queen_4_Y.4 (= 1 4) (= 2 Y.4))))
  (and
    (not (or
      (and Queen_1_Y.1 (= 1 1) (= 1 Y.1))
      (and Queen_2_Y.2 (= 1 2) (= 1 Y.2))
      (and Queen_3_Y.3 (= 1 3) (= 1 Y.3))
      (and Queen_4_Y.4 (= 1 4) (= 1 Y.4))))
    (not (or
      (and Queen_1_Y.1 (= 1 1) (= 3 Y.1))
      (and Queen_2_Y.2 (= 1 2) (= 3 Y.2))
      (and Queen_3_Y.3 (= 1 3) (= 3 Y.3))
      (and Queen_4_Y.4 (= 1 4) (= 3 Y.4))))
    (not (or
      (and Queen_1_Y.1 (= 1 1) (= 4 Y.1))
      (and Queen_2_Y.2 (= 1 2) (= 4 Y.2))
```

```
      (and Queen_3_Y.3 (= 1 3) (= 4 Y.3))
      (and Queen_4_Y.4 (= 1 4) (= 4 Y.4))))))
```

The output of the SMT solver is exactly as follows.

```
sat

MODEL
(= Queen_4_Y.4 true)
(= Queen_3_Y.3 true)
(= Queen_2_Y.2 true)
(= Queen_1_Y.1 true)
(= Y.4 3)
(= Y.2 4)
(= Y.3 1)
(= Y.1 2)
----
```

The propositional variable `Queen_1_Y.1` represents `Queen`$(1, $`Y.1`$)$, and since `Y.1` $= 2$, we have `Queen`$(1, 2)$. Similarly, the positions of all the queens can be read from the `Y` variables: $\{(1, 2), (2, 4), (3, 1), (4, 3)\}$.

7.2 Hydraulic Planning

The following is the formula for the problem of hydraulic planning in the notation of the grounder.

```
(
  (GIVEN
    (TYPES (TYPE INum))
    (SYMBOLS
      (SYMBOL A(INum INum))
      (SYMBOL P(INum INum INum))
      (SYMBOL T(INum))
      (SYMBOL C(INum INum))
      (SYMBOL O(INum INum))
      (SYMBOL G(INum))
```

```
    (SYMBOL V(INum))
    (SYMBOL N(INum))
  )
  (FUNCTIONS
    (FUNCTION _one():INum)
  )
)
(FIND P A)
(SATISFYING
  (FORALL x y i
    (OR
      (NOT
        (AND (V(x)) (V(y)) (N(i)))
      )
      (OR (O(x, y)) (C(x,y)) (NOT (P(x,y,i))))
    )
  )
  (FORALL x0 y0 x1 y1 i
    (OR
      (NOT
        (AND (C(x0,y0)) (C(x1,y1)) (N(i)))
      )
      (OR (NOT (P(x0,y0,i)))
        (NOT (P(x1,y1,i))) (AND (=(x0,x1)) (=(y0,y1)))
      )
    )
  )
  (FORALL x y i
    (OR
      (NOT
        (AND (O(x,y)) (N(i)))
      )
      (OR
        (NOT (P(x,y,i)))
        (T(x))
        (EXISTS j1 w1
          (AND (V(w1)) (<=(_one(),j1)) (<=(j1,i)) (P(w1,x,j1)))
```

```
          )
        )
      )
    )
    (FORALL x y i
      (OR
        (NOT
          (AND (C(x,y)) (N(i)))
        )
        (OR
          (NOT (P(x,y,i)))
          (T(x))
          (EXISTS j2 w2
            (AND (V(w2)) (<=(_one(),j2)) (<=(j2,i)) (P(w2,x,j2)))
          )
        )
      )
    )
    (FORALL y
      (OR
        (NOT (G(y)))
        (EXISTS x3 i3 (AND (N(i3)) (P(x3,y,i3))))
      )
    )
    (FORALL x y
      (OR
        (NOT (C(x,y)))
        (AND
          (OR (NOT (A(x,y))) (EXISTS i4 (AND (N(i4)) (P(x,y,i4)))))
          (OR (A(x,y)) (FORALL i (OR (NOT (N(i))) (NOT (P(x,y,i))))))
        )
      )
    )
  )
)
```

We will demonstrate the grounder using two instances. The first instance is as follows. This is the same instance as in the example in section 6.4.

```
(TYPE INum 5[1..5])
(PREDICATE V (1)(2)(3)(4)(5))
(PREDICATE N (1)(2))
(PREDICATE O (2,3)(1,4))
(PREDICATE C (1,2)(3,5)(4,5))
(PREDICATE T (1))
(PREDICATE G (5))
(FUNCTION _one (:1))
```

The output of the grounder when given this instance starts with a long list of things which P cannot contain, as in:

```
(not (or
  (and P_W1.2.3.1_2_J1.2.3.1 (= 1 W1.2.3.1) (= 1 2) (= 1 J1.2.3.1))
  (and P_W1.2.3.2_2_J1.2.3.2 (= 1 W1.2.3.2) (= 1 2) (= 1 J1.2.3.2))
  (and P_W1.2.3.3_2_J1.2.3.3 (= 1 W1.2.3.3) (= 1 2) (= 1 J1.2.3.3))
  (and P_W2.3.5.1_3_J2.3.5.1 (= 1 W2.3.5.1) (= 1 3) (= 1 J2.3.5.1))
  (and P_W2.4.5.1_4_J2.4.5.1 (= 1 W2.4.5.1) (= 1 4) (= 1 J2.4.5.1))
  (and P_W2.3.5.2_3_J2.3.5.2 (= 1 W2.3.5.2) (= 1 3) (= 1 J2.3.5.2))
  (and P_W2.4.5.2_4_J2.4.5.2 (= 1 W2.4.5.2) (= 1 4) (= 1 J2.4.5.2))
  (and P_W2.3.5.3_3_J2.3.5.3 (= 1 W2.3.5.3) (= 1 3) (= 1 J2.3.5.3))
  (and P_W2.4.5.3_4_J2.4.5.3 (= 1 W2.4.5.3) (= 1 4) (= 1 J2.4.5.3))
  (and P_X3.5_5_I3.5 (= 1 X3.5) (= 1 5) (= 1 I3.5))
  (and P_1_2_I4.1.2 (= 1 1) (= 1 2) (= 1 I4.1.2))
  (and P_3_5_I4.3.5 (= 1 3) (= 1 5) (= 1 I4.3.5))
  (and P_4_5_I4.4.5 (= 1 4) (= 1 5) (= 1 I4.4.5))))
(not (or
  (and P_W1.2.3.1_2_J1.2.3.1 (= 2 W1.2.3.1) (= 1 2) (= 1 J1.2.3.1))
  (and P_W1.2.3.2_2_J1.2.3.2 (= 2 W1.2.3.2) (= 1 2) (= 1 J1.2.3.2))
  (and P_W1.2.3.3_2_J1.2.3.3 (= 2 W1.2.3.3) (= 1 2) (= 1 J1.2.3.3))
  (and P_W2.3.5.1_3_J2.3.5.1 (= 2 W2.3.5.1) (= 1 3) (= 1 J2.3.5.1))
  (and P_W2.4.5.1_4_J2.4.5.1 (= 2 W2.4.5.1) (= 1 4) (= 1 J2.4.5.1))
  (and P_W2.3.5.2_3_J2.3.5.2 (= 2 W2.3.5.2) (= 1 3) (= 1 J2.3.5.2))
  (and P_W2.4.5.2_4_J2.4.5.2 (= 2 W2.4.5.2) (= 1 4) (= 1 J2.4.5.2))
  (and P_W2.3.5.3_3_J2.3.5.3 (= 2 W2.3.5.3) (= 1 3) (= 1 J2.3.5.3))
  (and P_W2.4.5.3_4_J2.4.5.3 (= 2 W2.4.5.3) (= 1 4) (= 1 J2.4.5.3))
  (and P_X3.5_5_I3.5 (= 2 X3.5) (= 1 5) (= 1 I3.5))
  (and P_1_2_I4.1.2 (= 2 1) (= 1 2) (= 1 I4.1.2))
```

```
  (and P_3_5_I4.3.5 (= 2 3) (= 1 5) (= 1 I4.3.5))
  (and P_4_5_I4.4.5 (= 2 4) (= 1 5) (= 1 I4.4.5))))
(not (or
  (and P_W1.2.3.1_2_J1.2.3.1 (= 3 W1.2.3.1) (= 1 2) (= 1 J1.2.3.1))
  (and P_W1.2.3.2_2_J1.2.3.2 (= 3 W1.2.3.2) (= 1 2) (= 1 J1.2.3.2))
  (and P_W1.2.3.3_2_J1.2.3.3 (= 3 W1.2.3.3) (= 1 2) (= 1 J1.2.3.3))
  (and P_W2.3.5.1_3_J2.3.5.1 (= 3 W2.3.5.1) (= 1 3) (= 1 J2.3.5.1))
  (and P_W2.4.5.1_4_J2.4.5.1 (= 3 W2.4.5.1) (= 1 4) (= 1 J2.4.5.1))
  (and P_W2.3.5.2_3_J2.3.5.2 (= 3 W2.3.5.2) (= 1 3) (= 1 J2.3.5.2))
  (and P_W2.4.5.2_4_J2.4.5.2 (= 3 W2.4.5.2) (= 1 4) (= 1 J2.4.5.2))
  (and P_W2.3.5.3_3_J2.3.5.3 (= 3 W2.3.5.3) (= 1 3) (= 1 J2.3.5.3))
  (and P_W2.4.5.3_4_J2.4.5.3 (= 3 W2.4.5.3) (= 1 4) (= 1 J2.4.5.3))
  (and P_X3.5_5_I3.5 (= 3 X3.5) (= 1 5) (= 1 I3.5))
  (and P_1_2_I4.1.2 (= 3 1) (= 1 2) (= 1 I4.1.2))
  (and P_3_5_I4.3.5 (= 3 3) (= 1 5) (= 1 I4.3.5))
  (and P_4_5_I4.4.5 (= 3 4) (= 1 5) (= 1 I4.4.5))))
```

Next, it lists every pair of valves that cannot both be opened in the same step. In other words, for every step, every pair of valves that are closed are not both opened in this step. For example, step 1, valves $(3, 5)$ and $(1, 2)$:

```
(or
  (not (or
    (and P_W1.2.3.1_2_J1.2.3.1 (= 3 W1.2.3.1) (= 5 2) (= 1 J1.2.3.1))
    (and P_W1.2.3.2_2_J1.2.3.2 (= 3 W1.2.3.2) (= 5 2) (= 1 J1.2.3.2))
    (and P_W1.2.3.3_2_J1.2.3.3 (= 3 W1.2.3.3) (= 5 2) (= 1 J1.2.3.3))
    (and P_W2.3.5.1_3_J2.3.5.1 (= 3 W2.3.5.1) (= 5 3) (= 1 J2.3.5.1))
    (and P_W2.4.5.1_4_J2.4.5.1 (= 3 W2.4.5.1) (= 5 4) (= 1 J2.4.5.1))
    (and P_W2.3.5.2_3_J2.3.5.2 (= 3 W2.3.5.2) (= 5 3) (= 1 J2.3.5.2))
    (and P_W2.4.5.2_4_J2.4.5.2 (= 3 W2.4.5.2) (= 5 4) (= 1 J2.4.5.2))
    (and P_W2.3.5.3_3_J2.3.5.3 (= 3 W2.3.5.3) (= 5 3) (= 1 J2.3.5.3))
    (and P_W2.4.5.3_4_J2.4.5.3 (= 3 W2.4.5.3) (= 5 4) (= 1 J2.4.5.3))
    (and P_X3.5_5_I3.5 (= 3 X3.5) (= 5 5) (= 1 I3.5))
    (and P_1_2_I4.1.2 (= 3 1) (= 5 2) (= 1 I4.1.2))
    (and P_3_5_I4.3.5 (= 3 3) (= 5 5) (= 1 I4.3.5))
    (and P_4_5_I4.4.5 (= 3 4) (= 5 5) (= 1 I4.4.5))))
  (not (or
    (and P_W1.2.3.1_2_J1.2.3.1 (= 1 W1.2.3.1) (= 2 2) (= 1 J1.2.3.1))
```

```
  (and P_W1.2.3.2_2_J1.2.3.2 (= 1 W1.2.3.2) (= 2 2) (= 1 J1.2.3.2))
  (and P_W1.2.3.3_2_J1.2.3.3 (= 1 W1.2.3.3) (= 2 2) (= 1 J1.2.3.3))
  (and P_W2.3.5.1_3_J2.3.5.1 (= 1 W2.3.5.1) (= 2 3) (= 1 J2.3.5.1))
  (and P_W2.4.5.1_4_J2.4.5.1 (= 1 W2.4.5.1) (= 2 4) (= 1 J2.4.5.1))
  (and P_W2.3.5.2_3_J2.3.5.2 (= 1 W2.3.5.2) (= 2 3) (= 1 J2.3.5.2))
  (and P_W2.4.5.2_4_J2.4.5.2 (= 1 W2.4.5.2) (= 2 4) (= 1 J2.4.5.2))
  (and P_W2.3.5.3_3_J2.3.5.3 (= 1 W2.3.5.3) (= 2 3) (= 1 J2.3.5.3))
  (and P_W2.4.5.3_4_J2.4.5.3 (= 1 W2.4.5.3) (= 2 4) (= 1 J2.4.5.3))
  (and P_X3.5_5_I3.5 (= 1 X3.5) (= 2 5) (= 1 I3.5))
  (and P_1_2_I4.1.2 (= 1 1) (= 2 2) (= 1 I4.1.2))
  (and P_3_5_I4.3.5 (= 1 3) (= 2 5) (= 1 I4.3.5))
  (and P_4_5_I4.4.5 (= 1 4) (= 2 5) (= 1 I4.4.5)))))
```

Next, we verify that each valve we open is pressurized before we open it. Since we are trying to supply pressure by opening a limited number of valves, we cannot afford to waste time opening valves without pressure. For example, before we open valve $(2, 3)$ in step 1, we make sure that some valve $(w, 2)$ has pressure in step 1 or earlier. Since there is no earlier step, we get:

```
(or
  (not (or
    (and P_W1.2.3.1_2_J1.2.3.1 (= 2 W1.2.3.1) (= 3 2) (= 1 J1.2.3.1))
    (and P_W1.2.3.2_2_J1.2.3.2 (= 2 W1.2.3.2) (= 3 2) (= 1 J1.2.3.2))
    (and P_W1.2.3.3_2_J1.2.3.3 (= 2 W1.2.3.3) (= 3 2) (= 1 J1.2.3.3))
    (and P_W2.3.5.1_3_J2.3.5.1 (= 2 W2.3.5.1) (= 3 3) (= 1 J2.3.5.1))
    (and P_W2.4.5.1_4_J2.4.5.1 (= 2 W2.4.5.1) (= 3 4) (= 1 J2.4.5.1))
    (and P_W2.3.5.2_3_J2.3.5.2 (= 2 W2.3.5.2) (= 3 3) (= 1 J2.3.5.2))
    (and P_W2.4.5.2_4_J2.4.5.2 (= 2 W2.4.5.2) (= 3 4) (= 1 J2.4.5.2))
    (and P_W2.3.5.3_3_J2.3.5.3 (= 2 W2.3.5.3) (= 3 3) (= 1 J2.3.5.3))
    (and P_W2.4.5.3_4_J2.4.5.3 (= 2 W2.4.5.3) (= 3 4) (= 1 J2.4.5.3))
    (and P_X3.5_5_I3.5 (= 2 X3.5) (= 3 5) (= 1 I3.5))
    (and P_1_2_I4.1.2 (= 2 1) (= 3 2) (= 1 I4.1.2))
    (and P_3_5_I4.3.5 (= 2 3) (= 3 5) (= 1 I4.3.5))
    (and P_4_5_I4.4.5 (= 2 4) (= 3 5) (= 1 I4.4.5))))
  (and
    (and
      (and
        (or
```

```
        (= W1.2.3.1 1)
        (= W1.2.3.1 2)
        (= W1.2.3.1 3)
        (= W1.2.3.1 4)
        (= W1.2.3.1 5))
      (<= 1 J1.2.3.1))
    (<= J1.2.3.1 1))
  P_W1.2.3.1_2_J1.2.3.1))
```

Now that we know that $P(x, y, i)$ is only true when valve (x, y) is open and has pressure at step i, we simply demand that the goal pipe, 5, is given pressure at some step i from some other pipe x. In other words, $P(x, 5, i)$, or as it appears in the grounder output:

```
(and (or (= I3.5 1) (= I3.5 2) (= I3.5 3)) P_X3.5_5_I3.5)
```

Finally, we need to ensure that A is the set of valves that are to be opened. Among the many things we test are the following, that $A(1, 2)$ implies $P(1, 2, i)$ for some i, and that $\neg A(1, 2)$ implies $\neg P(1, 2, i)$ for each of $i = 1$, $i = 2$, $i = 3$.

```
(or
  (not (or
    (and A_1_2 (= 1 1) (= 2 2))
    (and A_3_5 (= 1 3) (= 2 5))
    (and A_4_5 (= 1 4) (= 2 5))))
  (and (or (= I4.1.2 1) (= I4.1.2 2) (= I4.1.2 3)) P_1_2_I4.1.2))
(or A_1_2
  (and
    (not (or
      (and P_W1.2.3.1_2_J1.2.3.1 (= 1 W1.2.3.1) (= 2 2) (= 1 J1.2.3.1))
      (and P_W1.2.3.2_2_J1.2.3.2 (= 1 W1.2.3.2) (= 2 2) (= 1 J1.2.3.2))
      (and P_W1.2.3.3_2_J1.2.3.3 (= 1 W1.2.3.3) (= 2 2) (= 1 J1.2.3.3))
      (and P_W2.3.5.1_3_J2.3.5.1 (= 1 W2.3.5.1) (= 2 3) (= 1 J2.3.5.1))
      (and P_W2.4.5.1_4_J2.4.5.1 (= 1 W2.4.5.1) (= 2 4) (= 1 J2.4.5.1))
      (and P_W2.3.5.2_3_J2.3.5.2 (= 1 W2.3.5.2) (= 2 3) (= 1 J2.3.5.2))
      (and P_W2.4.5.2_4_J2.4.5.2 (= 1 W2.4.5.2) (= 2 4) (= 1 J2.4.5.2))
      (and P_W2.3.5.3_3_J2.3.5.3 (= 1 W2.3.5.3) (= 2 3) (= 1 J2.3.5.3))
      (and P_W2.4.5.3_4_J2.4.5.3 (= 1 W2.4.5.3) (= 2 4) (= 1 J2.4.5.3))
      (and P_X3.5_5_I3.5 (= 1 X3.5) (= 2 5) (= 1 I3.5))
      (and P_1_2_I4.1.2 (= 1 1) (= 2 2) (= 1 I4.1.2))
```

```
  (and P_3_5_I4.3.5 (= 1 3) (= 2 5) (= 1 I4.3.5))
  (and P_4_5_I4.4.5 (= 1 4) (= 2 5) (= 1 I4.4.5))))
(not (or
  (and P_W1.2.3.1_2_J1.2.3.1 (= 1 W1.2.3.1) (= 2 2) (= 2 J1.2.3.1))
  (and P_W1.2.3.2_2_J1.2.3.2 (= 1 W1.2.3.2) (= 2 2) (= 2 J1.2.3.2))
  (and P_W1.2.3.3_2_J1.2.3.3 (= 1 W1.2.3.3) (= 2 2) (= 2 J1.2.3.3))
  (and P_W2.3.5.1_3_J2.3.5.1 (= 1 W2.3.5.1) (= 2 3) (= 2 J2.3.5.1))
  (and P_W2.4.5.1_4_J2.4.5.1 (= 1 W2.4.5.1) (= 2 4) (= 2 J2.4.5.1))
  (and P_W2.3.5.2_3_J2.3.5.2 (= 1 W2.3.5.2) (= 2 3) (= 2 J2.3.5.2))
  (and P_W2.4.5.2_4_J2.4.5.2 (= 1 W2.4.5.2) (= 2 4) (= 2 J2.4.5.2))
  (and P_W2.3.5.3_3_J2.3.5.3 (= 1 W2.3.5.3) (= 2 3) (= 2 J2.3.5.3))
  (and P_W2.4.5.3_4_J2.4.5.3 (= 1 W2.4.5.3) (= 2 4) (= 2 J2.4.5.3))
  (and P_X3.5_5_I3.5 (= 1 X3.5) (= 2 5) (= 2 I3.5))
  (and P_1_2_I4.1.2 (= 1 1) (= 2 2) (= 2 I4.1.2))
  (and P_3_5_I4.3.5 (= 1 3) (= 2 5) (= 2 I4.3.5))
  (and P_4_5_I4.4.5 (= 1 4) (= 2 5) (= 2 I4.4.5))))
(not (or
  (and P_W1.2.3.1_2_J1.2.3.1 (= 1 W1.2.3.1) (= 2 2) (= 3 J1.2.3.1))
  (and P_W1.2.3.2_2_J1.2.3.2 (= 1 W1.2.3.2) (= 2 2) (= 3 J1.2.3.2))
  (and P_W1.2.3.3_2_J1.2.3.3 (= 1 W1.2.3.3) (= 2 2) (= 3 J1.2.3.3))
  (and P_W2.3.5.1_3_J2.3.5.1 (= 1 W2.3.5.1) (= 2 3) (= 3 J2.3.5.1))
  (and P_W2.4.5.1_4_J2.4.5.1 (= 1 W2.4.5.1) (= 2 4) (= 3 J2.4.5.1))
  (and P_W2.3.5.2_3_J2.3.5.2 (= 1 W2.3.5.2) (= 2 3) (= 3 J2.3.5.2))
  (and P_W2.4.5.2_4_J2.4.5.2 (= 1 W2.4.5.2) (= 2 4) (= 3 J2.4.5.2))
  (and P_W2.3.5.3_3_J2.3.5.3 (= 1 W2.3.5.3) (= 2 3) (= 3 J2.3.5.3))
  (and P_W2.4.5.3_4_J2.4.5.3 (= 1 W2.4.5.3) (= 2 4) (= 3 J2.4.5.3))
  (and P_X3.5_5_I3.5 (= 1 X3.5) (= 2 5) (= 3 I3.5))
  (and P_1_2_I4.1.2 (= 1 1) (= 2 2) (= 3 I4.1.2))
  (and P_3_5_I4.3.5 (= 1 3) (= 2 5) (= 3 I4.3.5))
  (and P_4_5_I4.4.5 (= 1 4) (= 2 5) (= 3 I4.4.5)))))))
```

The output of the SMT solver is as follows. The only interesting propositional variables are the ones that are true: `P_4_5_I4.4.5`, `P_X3.5_5_I3.5`, `P_W2.4.5.1_4_J2.4.5.1`, and `A_4_5`. They represent $P(4,5,1)$, $P(4,5,1)$, $P(1,5,1)$, and $A(4,5)$, which is a solution to the given instance.

```
sat
```

```
MODEL
(= A_1_2 false)
(= P_1_2_I4.1.2 false)
(= P_4_5_I4.4.5 true)
(= A_4_5 true)
(= P_W2.3.5.2_3_J2.3.5.2 false)
(= P_W1.2.3.1_2_J1.2.3.1 false)
(= P_W1.2.3.2_2_J1.2.3.2 false)
(= P_X3.5_5_I3.5 true)
(= P_W2.3.5.1_3_J2.3.5.1 false)
(= P_W2.4.5.2_4_J2.4.5.2 false)
(= A_3_5 false)
(= P_W2.4.5.1_4_J2.4.5.1 true)
(= P_3_5_I4.3.5 false)
(= J1.2.3.2 1)
(= J2.3.5.1 1)
(= W2.3.5.1 1)
(= X3.5 4)
(= J2.4.5.2 1)
(= J2.3.5.2 1)
(= W2.4.5.1 1)
(= W2.4.5.2 1)
(= I3.5 1)
(= I4.4.5 1)
(= W1.2.3.1 1)
(= W1.2.3.2 1)
(= I4.1.2 1)
(= J2.4.5.1 1)
(= I4.3.5 1)
(= J1.2.3.1 1)
(= W2.3.5.2 1)
----
```

To make the solution more elaborate, we will look at an instance with an additional vertex.

```
(TYPE INum 6[1..6])
(PREDICATE V (1)(2)(3)(4)(5)(6))
```

```
(PREDICATE N (1)(2)(3))
(PREDICATE O (2,3)(1,4))
(PREDICATE C (1,2)(3,5)(4,6))
(PREDICATE T (1))
(PREDICATE G (5)(6))
(FUNCTION _one (:1))
```

Since the output of the grounder is very similar in structure with this instance to what it was with the previous instance, we will not include any excerpts here. The output of the SMT solver is as follows.

```
sat

MODEL
(= P_W2.3.5.2_3_J2.3.5.2 true)
(= P_X3.6_6_I3.6 true)
(= P_W2.3.5.3_3_J2.3.5.3 false)
(= P_W2.3.5.1_3_J2.3.5.1 true)
(= A_1_2 true)
(= P_1_2_I4.1.2 true)
(= P_W2.4.6.1_4_J2.4.6.1 true)
(= P_W1.2.3.1_2_J1.2.3.1 true)
(= P_X3.5_5_I3.5 true)
(= P_W2.4.6.3_4_J2.4.6.3 true)
(= P_3_5_I4.3.5 true)
(= P_4_6_I4.4.6 true)
(= P_W2.4.6.2_4_J2.4.6.2 false)
(= P_W1.2.3.3_2_J1.2.3.3 false)
(= A_3_5 true)
(= P_W1.2.3.2_2_J1.2.3.2 false)
(= A_4_6 true)
(= J2.4.6.2 1)
(= W1.2.3.3 1)
(= J2.3.5.2 1)
(= W1.2.3.1 1)
(= W1.2.3.2 1)
(= W2.3.5.1 1)
(= W2.3.5.3 1)
```

```
(= I3.5 2)
(= X3.6 4)
(= J2.4.6.1 1)
(= I4.1.2 1)
(= J2.4.6.3 1)
(= J1.2.3.1 1)
(= J1.2.3.2 1)
(= J1.2.3.3 1)
(= X3.5 3)
(= I4.3.5 2)
(= W2.4.6.1 1)
(= J2.3.5.3 1)
(= I3.6 3)
(= J2.3.5.1 4)
(= W2.4.6.2 1)
(= I4.4.6 3)
(= W2.3.5.2 2)
(= W2.4.6.3 1)
----
```

From that output, the true propositional variables are:

Propositional Variable	Meaning
`P_W2.3.5.2_3_J2.3.5.2`	$P(2,3,1)$
`P_X3.6_6_I3.6`	$P(4,6,3)$
`P_W2.3.5.1_3_J2.3.5.1`	$P(1,3,4)$
`P_1_2_I4.1.2`	$P(1,2,1)$
`P_W2.4.6.1_4_J2.4.6.1`	$P(1,4,1)$
`P_W1.2.3.1_2_J1.2.3.1`	$P(1,2,1)$
`P_X3.5_5_I3.5`	$P(3,5,2)$
`P_W2.4.6.3_4_J2.4.6.3`	$P(1,4,1)$
`P_3_5_I4.3.5`	$P(3,5,2)$
`P_4_6_I4.4.6`	$P(4,6,3)$
`A_1_2`	$A(1,2)$
`A_3_5`	$A(3,5)$
`A_4_6`	$A(4,6)$

This result opens the valves in three steps, exactly as the instance requested.

7.3 Magic Square

For the purposes of timed testing, we will use the second version of the partial grounder, A, that allows expansion functions to be used in a theory file. This allows for simpler theory files and greatly improves the time performance of an SMT solver on the output of the partial grounder. The theory file a_1 that represents a magic square for A follows. It will be used to compare the time performance of an SMT solver on partial grounding with the time performance of a SAT solver on a grounding output by grounder B.

```
(
  (SATISFYING
    (FORALL x:IOrd y:IOrd (INum(f(x,y))))
    (FORALL x1:IOrd y1:IOrd x2:IOrd y2:IOrd
      (OR (AND (=(x1,x2)) (=(y1,y2))) (NOT (=(f(x1,y1),f(x2,y2))))))
    (FORALL y:IOrd (=(h(1,y),f(1,y))))
    (FORALL x:IOrd y:IOrd (=(h(x,y),+(h(-(x,1),y),f(x,y)))))
    (FORALL y:IOrd (=(h(n,y),c)))
    (FORALL x:IOrd (=(j(x,1),f(x,1))))
    (FORALL x:IOrd y:IOrd (=(j(x,y),+(j(x,-(y,1)),f(x,y)))))
    (FORALL x:IOrd (=(j(x,n),c)))
    (AND (=(k1(1),f(1,1))) (=(k2(1),f(n,1))))
    (FORALL i:IOrd (OR (=(i,1)) (=(k1(i),+(k1(-(i,1)),f(i,i))))))
    (FORALL i:IOrd (OR (=(i,1)) (=(k2(i),+(k2(-(i,1)),f(-(+(n,1),i),i))))))
    (AND (=(k1(n),c)) (=(k2(n),c)))
  )
)
```

The input syntax of A varies from the syntax used for first version of the partial grounder to allow for greater compatibility with the input language for B. The syntax is not completely compatible with the language for B, since the grounding solver does not allow expansion functions or numeral constants which have been allowed in A for clarity. The `INTEGER` function is also required by B but has been omitted from the requirements for A for clarity. Each variable following a quantifier is given a domain as is required by B, where `(FORALL x:P y:R (Q(x,y)))` represents $\forall x((P(x) \wedge R(y)) \supset Q(x, y))$.

For a 4x4 magic square, the instance for a_1 is as follows:

```
(TYPE IOrd 4[1..4])
(TYPE INum 16[1..16])
```

```
(FUNCTION n (:4))
```

The first line represents the predicate `IOrd` which represents the set of row and column numbers of the magic square, 1 through n. The second line represents the predicate `INum` which represents the set of numerals which may appear in the square, 1 through n^2. The last line represents the instance constant n, the number of the last row and last column of the magic square. The rule requiring a `_` before function and constant names has been omitted from A for clarity.

Since it is impossible to use expansion functions with B, an alternative axiomatization of magic square must be used. One possibility is to represent each expansion function as an expansion predicate, and axiomatize that each predicate represents a function. The following is axiomatization b_1 which represents this strategy in the input syntax of B.

```
(
  (GIVEN
    (TYPES (TYPE IOrd) (TYPE IOrdMax) (TYPE INum)
      (TYPE ISum) (TYPE ISumMax)
    )
    (SYMBOLS
      (SYMBOL F(IOrd IOrd INum))
      (SYMBOL H(IOrdMax IOrdMax ISumMax))
      (SYMBOL J(IOrdMax IOrdMax ISumMax))
      (SYMBOL K1(IOrdMax ISumMax))
      (SYMBOL K2(IOrdMax ISumMax))
    )
    (FUNCTIONS
      (FUNCTION _n():IOrd)
      (FUNCTION _one():IOrd)
    )
  )
  (FIND F H J K1 K2)
  (PHASES (PHASE (FIXPOINT )
  (SATISFYING
    (FORALL x:IOrd y:IOrd z1:INum z2:INum
      (OR (NOT (F(x,y,z1))) (NOT (F(x,y,z2))) (=(z1,z2))))
    (FORALL x:IOrdMax y:IOrdMax z1:ISumMax z2:ISumMax
      (OR (NOT (H(x,y,z1))) (NOT (H(x,y,z2))) (=(z1,z2))))
    (FORALL x:IOrdMax y:IOrdMax z1:ISumMax z2:ISumMax
      (OR (NOT (J(x,y,z1))) (NOT (J(x,y,z2))) (=(z1,z2))))
```

```
(FORALL i:IOrdMax z1:ISumMax z2:ISumMax
  (OR (NOT (K1(i,z1))) (NOT (K1(i,z2))) (=(z1,z2))))
(FORALL i:IOrdMax z1:ISumMax z2:ISumMax
  (OR (NOT (K2(i,z1))) (NOT (K2(i,z2))) (=(z1,z2))))

(FORALL x:IOrd y:IOrd (EXISTS z:INum (F(x,y,z))))
(FORALL x1:IOrd y1:IOrd x2:IOrd y2:IOrd z:INum
  (OR (NOT (F(x1,y1,z)))
    (AND (=(x1,x2)) (=(y1,y2))) (NOT (F(x2,y2,z)))
  )
)
(FORALL y:IOrd (EXISTS z:INum (AND (H(_one(),y,z)) (F(_one(),y,z)))))
(FORALL x:IOrd (EXISTS z:INum (AND (J(x,_one(),z)) (F(x,_one(),z)))))
(FORALL x:IOrd y:IOrd
  (OR (=(x,_one()))
    (EXISTS w:ISum z:INum
      (AND
        (H(-(INTEGER(x),INTEGER(_one())),y,w))
        (F(x,y,z))
        (H(x,y,+(INTEGER(w),INTEGER(z))))
      )
    )
  )
)
(FORALL x:IOrd y:IOrd
  (OR (=(y,_one()))
    (EXISTS w:ISum z:INum
      (AND
        (J(x,-(INTEGER(y),INTEGER(_one())),w))
        (F(x,y,z))
        (J(x,y,+(INTEGER(w),INTEGER(z))))
      )
    )
  )
)
(EXISTS c:ISum
  (AND
    (FORALL y:IOrd (H(_n(),y,c)))
    (FORALL x:IOrd (J(x,_n(),c)))
    (K1(_n(),c)) (K2(_n(),c))
```

```
      )
    )
    (EXISTS w:INum (AND (K1(_one(),w)) (F(_one(),_one(),w))))
    (EXISTS w:INum (AND (K2(_one(),w)) (F(_n(),_one(),w))))
    (FORALL i:IOrd
      (OR (=(i,_one()))
        (EXISTS w:ISum z:INum
          (AND
            (K1(-(INTEGER(i),INTEGER(_one())),w))
            (F(i,i,z))
            (K1(i,+(INTEGER(w),INTEGER(z))))
          )
        )
      )
    )
    (FORALL i:IOrd
      (OR (=(i,_one()))
        (EXISTS w:ISum z:INum
          (AND
            (K2(-(INTEGER(i),INTEGER(_one())),w))
            (F(-(+(INTEGER(_n()),INTEGER(_one())),INTEGER(i)),i,z))
            (K2(i,+(INTEGER(w),INTEGER(z))))
          )
        )
      )
    )
  ))) (PRINT (F))
)
```

The instance file for a 3x3 magic square with b_1 is as follows.

```
(TYPE IOrd 3[1..3])
(TYPE IOrdMax 4[0..3])
(TYPE INum 9[1..9])
(TYPE ISum 15[1..15])
(TYPE ISumMax 24[1..24])
(FUNCTION _n (:3))
(FUNCTION _one (:1))
```

The sort `IOrd` represents the set of numbers of the rows and columns of the magic square.

The sort `IOrdMax` represents the range of row and column numbers that may occur in the grounding, which is the set `IOrd` plus 0 which may occur when subtracting one from a row or column number. The sort `INum` represents the set of numerals which may be contained in the magic square. The sort `ISum` represents a set containing at least all possible sums of elements of the rows, columns, or diagonals in a correct magic square. For a 3x3 magic square this happens to be 1 through 15 because the sum of all rows, columns, and diagonals must be $n(n^2+1)/2$, so any subset of elements from any particular row, column, or diagonal must be at most 15. The sort `ISumMax` represents a set containing at least all of `ISum`, plus the sum of each element of `INum` with each element of `ISum`. This is required so that the grounder can add elements of `ISum` and elements of `INum`. The constant `_n` represents the number of the last row and column of the square. The constant `_one` represents the number 1 because B does not permit numeral constants.

The grounder B happens to provide an alternate way to axiomatize magic square that performs much better than b_1, but uses an entirely different axiomatization. The `SUM` operator allows input to B to express magic squares much more compactly. The following axiomatization is b_2, the magic square axiomatization for B using `SUM`. The `SUM` operator is used as a term written `(SUM(` $\bar{x}$ `;` $y(\bar{x})$ `;` $z(\bar{x})$ `)`, $\bar{x}$ is a list of variables with domains, $y(\bar{x})$ is a term, and $z(\bar{x})$ is a formula. The value of the `SUM` term is the sum of $y(\bar{x})$ for all $\bar{x}$ in the given domain such that $z(\bar{x})$ is satisfied.

```
(
  (GIVEN
    (TYPES (TYPE IOrd) (TYPE INum) (TYPE ISum))
    (SYMBOLS
      (SYMBOL F(IOrd IOrd INum))
    )
    (FUNCTIONS
      (FUNCTION _n():IOrd)
      (FUNCTION _c():ISum)
      (FUNCTION _one():IOrd)
    )
  )
  (FIND F)
  (PHASES (PHASE (FIXPOINT )
  (SATISFYING
    (FORALL x:IOrd y:IOrd z1:INum z2:INum
```

```
      (OR (NOT (F(x,y,z1))) (NOT (F(x,y,z2))) (=(z1,z2))))
    (FORALL x:IOrd y:IOrd (EXISTS z:INum (F(x,y,z))))
    (FORALL x1:IOrd y1:IOrd x2:IOrd y2:IOrd z:INum
      (OR (NOT (F(x1,y1,z)))
         (AND (=(x1,x2)) (=(y1,y2))) (NOT (F(x2,y2,z)))
      )
    )
    (FORALL x:IOrd
      (=(INTEGER(_c()),INTEGER(SUM(y:IOrd z:INum;z;(F(x,y,z))))))
    )
    (FORALL y:IOrd
      (=(INTEGER(_c()),INTEGER(SUM(x:IOrd z:INum;z;(F(x,y,z))))))
    )
    (=(INTEGER(_c()),
      INTEGER(SUM(i:IOrd z:INum;z;(F(i,i,z)))))
    )
    (=(INTEGER(_c()),
      INTEGER(
        SUM(
          i:IOrd z:INum;z;
          (F(-(+(INTEGER(_n()),INTEGER(_one())),INTEGER(i)),i,z)))
        )
      )
    )
  ))) (PRINT (F))
)
```

The sum of each row, column, and diagonal is called `_c` in this axiomatization. For grounding each variable must be given a domain, but there is no natural domain for `_c`. By studying magic squares in general, we know that for any n, the correct value for `_c` is $n(n^2+1)/2$, and any domain containing that value would allow the solver to find a magic square. Since we have no way to prefer one domain over another, we will choose the domain that makes solving fastest, the domain containing only $n(n^2+1)/2$, by making `_c` an instance constant equal to $n(n^2+1)/2$.

The following is the instance for a 5x5 magic square with b_2.

```
(TYPE IOrd 5[1..5])
```

```
(TYPE INum 25[1..25])
(TYPE ISum 1[65..65])
(FUNCTION _n (:5))
(FUNCTION _one (:1))
(FUNCTION _c (:65))
```

Table 7.1 provides a comparison of solving times when the output of a grounder or partial grounder is given to an appropriate solver. The axioms a_1 is partially ground and given to an SMT solver, while b_1 and b_2 are ground and given to a SAT solver. The grounding time is not included. The result is left blank when the solver runs for more than an hour, since it is assumed to be impractical for finding magic squares at that point and beyond. The axioms b_2 caused a segmentation fault in the solver for the 6x6 instance, so it failed to return a result in any amount of time.

Size	3 x 3	4 x 4	5 x 5	6 x 6
a_1	0.0 seconds	0.0 seconds	7 minutes	
b_1	0.9 seconds	20 minutes		
b_2	0.0 seconds	20 seconds	7 minutes	Segmentation fault

Table 7.1: Approximate solving time for magic squares by axioms and instances

Chapter 8

Conclusion

8.1 Usefulness of Reduction to SMT

Satisfiability Modulo Theories solvers are powerful and flexible tools for solving a wide variety of problems efficiently, but a common practice of making solvers that do not allow universal quantifiers results in many very large SMT formulas that are more meaningful to a machine than to a person, a problem that is shared with SAT solvers. In order to make an SMT solver into a truly convenient tool, there must be an additional tool that can assist with the construction of an SMT formula.

Using a partial grounder as a preprocessor, the universal quantifier becomes a macro that can generate an SMT formula of any size from single small and intuitive input formula. Instead of writing fifty variations of $Q(y, z) \wedge y + 1 < z$ by hand, we can write $\forall x(P(x) \supset \exists yz(Q(y, z) \wedge y + x < z))$ just once, and write an instance structure to define $P = \{1, \ldots, 50\}$.

As a very useful tool, a partial grounder should be included with every SMT solver.

8.2 Future Software

It should be simple to implement a summation operator for a partial grounder to SMT. By using a non-first-order term of the form $\texttt{SUM}(\bar{x};\alpha(\bar{x});\phi(\bar{x}))$, where $\bar{x}$ is a tuple of variables, $\alpha(\bar{x})$ is a term, and $\phi(\bar{x})$ is a formula, one can construct a sum $a_1 + \ldots + a_n$, with additional formula to ensure that $a_1 = \alpha(\bar{y}_1)$ if $\phi(\bar{x})$ is true, and $a_1 = 0$ otherwise.

To produce the same effect without a SUM operator tends to require an additional expansion function f such that $f(i)$ represents the sum up to index i and $(\phi(\bar{y}_i) \supset f(i + 1) =$

$f(i) + \alpha(\bar{y}_i)) \wedge (\neg\phi(\bar{y}_i) \supset f(i+1) = f(i))$. In the first version of the partial grounder, f would have to be represented by a predicate, adding even more to ground, but even in the second version of the partial grounder the resulting SMT formula far larger than the simple $a_1 + \ldots + a_n$ that is intended. Experiments should be done with `SUM` to get possibly even better performance from the SMT solver.

When using an satisfiability modulo theories solver it is necessary to convert the output of the solver into a form that more directly expresses the solution to the original model expansion problem. There is no standardized output format for all SMT solvers in the way that the SMT-LIB standard provides a uniform notation for inputs, but it would be invaluable to have a tool to do the conversion automatically for whatever SMT solver is being used. It is time-consuming and error-prone to do it by hand even for small examples.

Bibliography

[1] A. Aavani, S. Tasharrofi, G. Unel, E. Ternovska, and D. Mitchell. Speed-up techniques for negation in grounding. In *Proc., 16th Int'l Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-16)*, LNCS, 2010.

[2] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proc. 23rd ACM Symp. on the Theory of Computing*, pages 209–219, 1991.

[3] H. Andréka, J. Van Benthem, and I. Németi. Modal languages and bounded fragments of predicate logic. *J. Phil. Logic*, 49(3):217–274, 1998.

[4] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[5] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[6] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010.

[7] D. M. Barrington, N. Immerman, and H. Straubing. On uniformity within NC^1. *Journal of Computer and System Sciences*, 41(3):274 – 306, 1990.

[8] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT solver. In *Proc. CAV '08*, pages 299–303, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all problems in NP. *Computer Languages*, 26:165–195, 2000.

[10] M. Cadoli and T. Mancini. Combining Relational Algebra, SQL, and constraint programming. In *Proc., FroCos-02*, pages 147–161, 2002.

[11] M. Cadoli and T. Mancini. Automated reformulation of specifications by safe delay of constraints. *Artificial Int.*, 170(8–9):779–801, 2006.

[12] M. Cadoli, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all problems in NP. In *Proc., First Int'l Workshop on Practical Aspects of Declarative Languages (PADL)*, pages 16–30, 1999.

[13] M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. *Artificial Intelligence*, 162:89–120, 2005.

[14] A. Chandra and D. Harel. Computable queries for relational databases. *Journal of Computer and System Sciences*, 21:156–178, 1980.

[15] S. Cook. Theories for complexity classes and their propositional translations. In Jan Krajicek, editor, *Complexity of computations and proofs*, pages 175–227. Quaderni di Matematica, 2003.

[16] S. A. Cook. A hierarchy for nondeterministic time complexity. *Journal of Computer and System Sciences*, 7(4):343–353, 1973.

[17] S.A. Cook and A. Kolokolova. A second-order system for polynomial-time reasoning based on Grädel's theorem. In *Proceedings of the Sixteens annual IEEE symposium on Logic in Computer Science*, pages 177–186, 2001.

[18] M. Dao-Tran, T. Eiter, M. Fink, and T. Krennwallner. Distributed nonmonotonic multi-context systems. In *Proc. KR'10*.

[19] L. de Moura and N. Bjørner. Model-based theory combination. *Electron. Notes Theor. Comput. Sci.*, 198(2):37–49, 2008.

[20] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. 4963:337–340, 2008. 10.1007/978-3-540-78800-3_24.

[21] M. Denecker. Extending classical logic with inductive definitions. In *Proc., First Int'l Conference on Computational Logic (CL-2000)*, pages 703–717. Springer, 2000.

[22] M. Denecker and E. Ternovska. Inductive situation calculus. In *Proc., Ninth Int'l Conf. on Principles of Knowledge Representation and Reasoning (KR-04)*, pages 545–553. AAAI Press, 2004.

[23] M. Denecker and E. Ternovska. A logic of non-monotone inductive definitions and its modularity properties. In *Proc., 7th Int'l Conf., on Logic Programming and Non-monotonic Reasoning (LPNMR-04)*, pages 47–60. Springer, 2004. LNCS 2923.

[24] M. Denecker and E. Ternovska. Inductive situation calculus. *Artificial Intelligence*, 171(5-6):332–360, 2007.

[25] M. Denecker and E. Ternovska. A logic of non-monotone inductive definitions. *ACM transactions on computational logic (TOCL)*, 9(2):1–51, 2008.

[26] B. Dutertre and L. de Moura. The Yices SMT solver. August 2006.

[27] D. East and M. Truszczynski. Predicate-calculus based logics for modeling and solving search problems. *ACM Transactions on Computational Logic (TOCL)*, 7(1):38–83, 2006.

[28] H. D. Ebbinghaus and J. Flum. *Finite model theory*. Springer Verlag, 1995.

[29] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In *Complexity of Computation*, volume 7, pages 43–73, 1974.

[30] R. Fagin. Finite-model theory – a personal perspective. *Theoretical Computer Science*, 116:3–31, 1993.

[31] P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *Proc., 9th Int'l Conf. on Principles and Practice of Constraint Programming (CP-03)*, page 971. Springer, 2003. LNCS 2833.

[32] H. M. Friedman. Some decision problems of enormous complexity. In *Proc., LICS'99*, pages 2–13, 1999.

[33] A. M. Frisch, M. Grum, C. Jefferson, B. M. Hernandez, and I. Miguel. The design of ESSENCE: a constraint language for specifying combinatorial problems. In *Proc. IJCAI'07*, 2007.

[34] A. M. Frisch, W. Harvey, C. Jefferson, B. M. Hernández, and I. Miguel. ESSENCE: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

[35] M. J. García de la Banda, K. Marriott, R. Rafeh, and M. Wallace. The modelling language zinc. In *CP*, pages 700–705, 2006.

[36] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental ASP solver. In *Proc. ICLP'08*, volume 5366 of *LNCS*, pages 190–205.

[37] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[38] G. Gottlob, E. Grädel, and H. Veith. Datalog LITE: a deductive query language with linear time model checking. *ACM Trans. Comput. Logic (TOCL)*, 3(1):42–79, 2002.

[39] G. Gottlob, N. Leone, and F. Scarcello. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. In *Proc., Twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS-01)*, pages 195–206. ACM, 2001.

[40] E. Grädel. The Expressive Power of Second Order Horn Logic. In *Proceedings of 8th Symposium on Theoretical Aspects of Computer Science STACS '91, Hamburg 1991*, volume 480 of *LNCS*, pages 466–477. Springer-Verlag, 1991.

[41] E. Grädel. Capturing Complexity Classes by Fragments of Second Order Logic. *Theor. Comp. Sc.*, 101:35–57, 1992.

[42] E. Gradel. Guarded fragments of first-order logic: A perspective for new description logics? In *In Proc. of 1998 Int. Workshop on Description Logics DL '98, Trento, CEUR Electronic Workshop Proceedings*, pages 1–1. Publications, 1998.

[43] E. Grädel. The decidability of guarded fixed point logic. In J. Gerbrandy, M. Marx, M. de Rijke, and Y. Venema, editors, *JFAK. Essays Dedicated to Johan van Benthem on the Occasion of his 50th Birthday*. Amsterdam University Press, 1999.

[44] E. Grädel. On the restraining power of guards. *Journal of Symbolic Logic*, 64:1719–1742, 1999.

[45] E. Grädel. Guarded fixed point logic and the monadic theory of trees. *Theoretical Computer Science*, 288:129–152, 2002.

[46] E. Grädel. Finite model theory and descriptive complexity. In *Finite Model Theory and Its Applications*, pages 125–230. Springer, 2007.

[47] E. Grädel and Y. Gurevich. Metafinite model theory. *Information and Computation*, 140(1):26–81, 1998.

[48] E. Grädel, C. Hirsch, and M. Otto. Back and forth between guarded and modal logics. *ACM Trans. Comput. Logic*, 3(3):418–463, 2002.

[49] E. Grädel and M. Otto. On logics with two variables. *Theoretical Computer Science*, 224:73–113, 1999.

[50] E. Grädel and I. Walukiewicz. Guarded fixed point logic. In *LICS'99*, pages 45–55, 1999.

[51] E. Graedel, P. G. Kolaitis, L. Libkin, M. Marx, J. Spencer, M.Y. Vardi, Y. Venema, and S. Weinstein. *Finite Model Theory and Applications*. Springer, 2007.

[52] C. Green. Theorem proving by resolution as a basis for question-answering systems. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 183–205. Edinburgh University Press, 1969.

[53] S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Guarded open answer set programming with generalized literals. In *Proc., FoIKS*, pages 179–200, 2006.

[54] J. Hooker. *Logic-based methods for optimization: combining optimization and constraint satisfaction*, chapter 19, pages 389–422. Wiley and Sons, 2000.

[55] N. Immerman. Relational queries computable in polytime. In *14th ACM Symp. on Theory of Computing (STOC)*, pages 147 –152. Springer Verlag, 1982.

[56] N. Immerman. Languages that capture complexity classes. In *15th ACM STOC symposium*, pages 347–354, 1983.

[57] N. Immerman. Relational queries computable in polytime. *Information and Control*, 68:86 –104, 1986.

[58] N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16(4):760–778, 1987.

[59] N. Immerman. *Descriptive complexity*. Springer Verlag, New York, 1999.

[60] T. Janhunen, E. Oikarinen, H. Tompits, and S. Woltran. Modularity aspects of disjunctive stable models. In C. Baral, G. Brewka, and J. Schlipf, editors, *the Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *Lecture Notes in Artificial Intelligence*, pages 175–187. Springer-Verlag, 2007.

[61] M. Järvisalo, E. Oikarinen, T. Janhunen, and I. Niemelä. A module-based framework for multi-language constraint modeling. In *Proc. LPNMR'09*, volume 5753 of *LNCS*.

[62] P. G. Kolaitis and M. N. Thakur. Logical definability of NP optimization problems. *Information and Computation*, 50(3):321–353, 1994.

[63] P. G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 205–213, 1998.

[64] A. Kolokolova. *Systems of bounded arithmetic from descriptive complexity*. PhD thesis, University of Toronto, October 2004.

[65] A. Kolokolova. Closure properties of weak systems of bounded arithmetic. In *Computer Science Logic: 19th International Workshop. Proceedings*, volume 3634 of *LNCS*, pages 369–383, 2005.

[66] A. Kolokolova, Y. Liu, D. Mitchell, and E. Ternovska. On the complexity of model expansion. In *Proce. LPAR-17*, ARCoSS subline of LNCS. Springer, 2010.

[67] A. Kolokolova, Y. Liu, D. Mitchell, and E. Ternovska. On the complexity of model expansion. In *Proc., 17th Int'l Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-17)*, pages 447–458. Springer, 2010. LNCS 6397.

[68] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3), 2006.

[69] H. J. Levesque. Making believers out of computers. *Artif. Intell.*, 30:81–108, October 1986.

[70] H. J. Levesque and R. J. Brachman. Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence*, 3:78–93, 1987.

[71] H. J. Levesque and G. Lakemeyer. *The Logic of Knowledge Bases*. The MIT Press, 2001.

[72] L. Libkin. *Elements of Finite Model Theory*. Springer Verlag, 2004.

[73] L. Libkin. *Embedded Finite Models and Constraint Databases*, pages 257–338. Springer, 2007.

[74] Y. Liu and H. J. Levesque. A tractability result for reasoning with incomplete first-order knowledge bases. In *Proc. of the 18th Int. Joint Conf. on Artif. Intell. (IJCAI)*, pages 83–88, 2003.

[75] A. Livchak. The relational model for process control. (in russian). *Automated Documentation and Mathematical Linguistics*, (4):27–29, 1983.

[76] A.B. Livchak. Languages for polynomial-time queries. In *Computer-based modeling and optimization of heat-power and electrochemical objects*, page 41, 1982.

[77] T. Mancini. *Declarative constraint modelling and specification-level reasoning*. PhD thesis, 2005.

[78] T. Mancini and M. Cadoli. Exploiting functional dependencies in declarative problem specifications. *Artificial Intelligence*, 171(16–17):985–1010, 2007.

[79] V. W. Marek and J. B. Remmel. On the expressibility of stable logic programming. *TCLP*, 3:551, 2003.

[80] V. W. Marek and M. Truszczynski. *Stable logic programming - an alternative logic programming paradigm*, pages 375–398. Springer-Verlag, 1999. In: The Logic Programming Paradigm: A 25-Year Perspective, K.R. Apt, V.W. Marek, M. Truszczynski, D.S. Warren, Eds.

[81] M. Mariën, R. Mitra, M. Denecker, and M. Bruynooghe. Satisfiability checking for PC(ID). In *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2005, Proceedings*, volume 3835 of *Lecture Notes in Computer Science*, pages 565–579. Springer, 2005.

[82] M. Mariën, J. Wittocx, and M. Denecker. The IDP framework for declarative problem solving. In *Search and Logic: Answer Set Programming and SAT*, pages 19–34, 2006.

[83] K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. Garcia de la Banda, and M. Wallace. The design of the zinc modelling language. *Constraints*, 13(3):229–267, 2008.

[84] J. A. Medina and N. Immerman. A syntactic characterization of NP-completeness. In *Proceedings of Logic in Computer Science 1994 (LICS'94)*, pages 241–250, 1994.

[85] V. Mellarkod, M. Gelfond, and Y. Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53:251–287, 2008. 10.1007/s10472-009-9116-y.

[86] D. Mitchell and E. Ternovska. Constraint programming with unrestricted quantification. In *Proc. First International Workshop on Quantification in Constraint Programming (CP-2005 Workshop)*, 2005.

[87] D. Mitchell and E. Ternovska. A framework for representing and solving NP search problems. In *Proc. of the 20th National Conf. on Artif. Intell. (AAAI)*, pages 430–435, 2005.

[88] D. Mitchell, E. Ternovska, F. Hach, and R. Mohebali. Model expansion as a framework for modelling and solving search problems. Technical Report TR 2006-24, Simon Fraser University, School of Computing Science, 2006.

[89] D. G. Mitchell, F. Hach, and R. Mohebali. Faster phylogenetic inferernce with MXG. In *Proc. LPAR'07*, 2007.

[90] D. G. Mitchell and E. Ternovska. On the expressive power of ESSENCE. In *Workshop Proceedings, ModRef'07*, 2007.

[91] D. G. Mitchell and E. Ternovska. Expressiveness and abstraction in ESSENCE. *Constraints*, 13(2):343–384, 2008.

[92] R. Mohebali, F. Hach, and D. G. Mitchell. MXG: a model expansion grounder and solver, 2007. LPAR 2007 short paper.

[93] I. Niemela. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.

[94] I. Niemelä. Integrating answer set programming and satisfiability modulo theories. In *Proc. LPNMR '09*, pages 3–3, Berlin, Heidelberg, 2009. Springer-Verlag.

[95] E. Oikarinen and T. Janhunen. Modular equivalence for normal logic programs. In J. Dix and A. Hunter, editors, *the Proceedings of the 11th Workshop on Nonmonotonic Reasoning (NMR'06)*, pages 10–18, 2006.

[96] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[97] M. Patterson, Y. Liu, E. Ternovska, and A. Gupta. Grounding for model expansion in k-guarded formulas, 2006. Short presentation at 21st IEEE Symposium on Logic in Computer Science (LICS).

[98] M. Patterson, Y. Liu, E. Ternovska, and A. Gupta. Grounding for model expansion in k-guarded formulas with inductive definitions. In *Proc. IJCAI'07*, 2007.

[99] N. Pelov and E. Ternovska. Reducing inductive definitions to propositional satisfiability. In *Proc., ICLP-05*, pages 221–234, 2005.

[100] K. Pipatsrisawat and A. Darwiche. On decomposability and interaction functions. In *Proc. ECAI'10*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 9–14. IOS Press.

[101] K. Pipatsrisawat and A. Darwiche. Top-down algorithms for constructing structured DNNF: Theoretical and practical implications. In *Proc. ECAI'10*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 3–8. IOS Press.

[102] A. Rrustemi. The application of sat-solvers in combinatorial problems, 2004. A dissertation submitted for the degree Diploma in Computer Science.

[103] R. Sebastiani. Lazy satisfiability modulo theories. Journal on Satisfiability, Boolean Modeling and Computation, J.SAT 3, 2007.

[104] L. Stockmeyer. *The Complexity of Decision Problems in Automata Theory.* PhD thesis, MIT, 1974.

[105] L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1977.

[106] D. Suciu. Domain-independent queries on databases with external functions. *Theoretical Computer Science*, 190(2):279–315, 1998.

[107] S. Tasharrofi and E. Ternovska. Built-in arithmetic in knowledge representation languages. In *Proceedings of NonMon@30, Thirty Years of Nonmonotonic Reasoning*, 2010.

[108] S. Tasharrofi and E. Ternovska. Built-in arithmetic in knowledge representation languages. In *NonMon at 30 (Thirty Years of Nonmonotonic Reasoning)*, October 2010.

[109] S. Tasharrofi and E. Ternovska. PBINT, a logic for modelling search problems involving arithmetic. In Christian Fermüller and Andrei Voronkov, editors, *Proceedings of Logic for Programming, Artificial Intelligence and Reasoning (LPAR 17)*, ARCoSS subline of LNCS. Springer, 2010.

[110] E. Ternovska and D. Mitchell. Declarative programming of search problems declarative programming of search problems with built-in arithmetic. In *Proceedings of IJCAI 2009*, pages 942–947, 2009.

[111] E. Ternovska and D. G. Mitchell. Declarative programming of search problems with built-in arithmetic. Technical Report TR-2007-28, Simon Fraser University, 2007.

[112] E. Ternovskaia. Causality via inductive definitions. In Charles L. Ortiz, Jr., editor, *Working Notes of the AAAI Spring Symposium on Prospects for a Commonsense Theory of Causation*, pages 94–100, Menlo Park, CA, 1998.

[113] E. Ternovskaia. Inductive definability and the situation calculus. *Lecture Notes in Computer Science*, 1472:227–248, 1998.

[114] B. Trahtenbrot. The impossibility of an algorithm for the decision problem for finite domains. *Doklady Academii Nauk SSSR*, 70:569–572, 1950.

[115] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of Assoc. Comput. Mach.*, 38(3):620–650, 1991.

[116] P. Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, 1999.

[117] M. Vardi. Complexity of relational query languages. *Information and Control*, 68:137 –146, 1986.

[118] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *14th ACM Symp.on Theory of Computing, Springer Verlag*, pages 137–146, 1982.

[119] M. Y. Vardi. On the complexity of bounded-variable queries. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California*, pages 266–276. ACM Press, 1995.

[120] J. Wittocx and M. Marien. *The IDP System*. Katholieke Universiteit Leuven, June 2008. (Manual at www.cs.kuleuven.be/~dtai/krr/software/idpmanual.pdf).