A METHOD FOR SOLVING NP SEARCH BASED ON MODEL EXPANSION AND GROUNDING

by

Raheleh Mohebali
B.Sc., University of Tehran, 2003

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

APPROVAL

Name:	Raheleh Mohebali
Degree:	Master of Science
Title of thesis:	A Method for Solving NP Search Based on Model Expansion and Grounding

Examining Committee: Dr. Andrei Bulatov
Chair

____

Dr. David G. Mitchell, Assistant Professor, Computing Science
Simon Fraser University
Senior Supervisor

____

Dr. Eugenia Ternovska, Assistant Professor, Computing Science
Simon Fraser University
Supervisor

____

Dr. William S. Havens, Associate Professor, Computing Science
Simon Fraser University
SFU Examiner

Date Approved: Dec 6, 2007

Abstract

The logical task of model expansion (MX) has been proposed as a declarative constraint programming framework for solving search and decision problems. We present a method for solving NP search problems based on MX for first order logic extended with inductive definitions and cardinality constraints. The method involves grounding, and execution of a propositional solver, such as a SAT solver. Our grounding algorithm applies a generalization of the relational algebra to construct a ground formula representing the solutions to an instance. We demonstrate the practical feasibility of our method with an implementation, called MXG. We present axiomatizations of several NP-complete benchmark problems, and experimental results comparing the performance of MXG with state-of-the-art Answer Set programming (ASP) solvers. The performance of MXG is competitive with, and often better than, the ASP solvers on the problems studied.

To my parents

Acknowledgments

I would like to thank my senior supervisor, Dr. David Mitchell, for many insightful conversations during the development of the ideas in this thesis, and for helpful comments on the text. I would also like to thank Dr. Eugenia Ternovska and Dr. William Havens for providing me with valuable comments on the final document. I also thank Nhan Nguyen for imeplementation of a minimum-Horn model module in MXG.

Contents

Approval ii

Abstract iii

Dedication iv

Acknowledgments v

Contents vi

List of Tables ix

List of Figures x

1 Introduction 1

2 MXG Overview 5

2.1 MX Mathematical Overview 5

2.2 MXG: A FO-MX Solver 8

2.3 MXG Solving Approach 9

2.4 Extensions to First Order Logic 10

2.4.1 Multi-sorted Logic 10

2.4.2 Order 11

2.4.3 Inductive Definitions 11

2.4.4 Cardinality Constraints 14

3 MXG Input Language **16**
3.1 Instance Description Syntax 17
3.2 Problem Specification Syntax 18
3.2.1 Given Section 18
3.2.2 Find Section 19
3.2.3 Satisfying Section 19

4 MXG Solving Method **22**
4.1 "Grounder" Module 22
4.1.1 Converting a Reduced Ground FO Formula to Clausal Form 23
4.1.2 Transforming a FO Clause to a Propositional Clause 24
4.2 "Un-grounder" module 26
4.3 Handling Inductive Definitions 26
4.3.1 Horn-ID 27
4.3.2 Comp-ID 27
4.4 Handling Cardinality Constraints 27
4.5 Reducing the Number of Tseitin Variables 29

5 A Grounding Algorithm for MXG **31**
5.1 Gnd: A Grounding with Extended Relations 31
5.1.1 An algebra for extended relations 32
5.1.2 Algorithm Gnd 35
5.2 Gnd-Hidden: Grounding With Hidden Variables 37
5.2.1 An Algebra For Extended Hidden Relations 38
5.2.2 Gnd-Hidden Algorithm 40
5.3 Complexity of Gnd-Hidden Algorithm 41
5.4 Join Refinement by Indexing 42

6 Evaluation **43**
6.1 Comparing Grounding Times of MXG and Lparse 45
6.2 Overall Solving Performance 46
6.2.1 Cumulative Performance Plots 46
6.2.2 Test Platform 47
6.3 Graph K-Colouring 47

6.4 Latin Square Completion 48
6.5 Blocked Queens 50
6.6 Social Golfer 51
6.7 Bounded Spanning Tree 53

7 Related Work **55**
7.1 Other FO(ID)-MX Solvers 55
7.2 ASP Systems 56
7.3 Constraint Modelling Languages 57
7.4 Finite Model Finders (Generators) 57

8 Conclusion and Future work **60**
8.1 Future Work 61

A MXG Problem Specification Grammar **62**

B MXG Instance Description Grammar **64**

Bibliography **65**

List of Tables

3.1 ASCII Equivalents for Logical Symbols 20

6.1 Grounding time in seconds of MXG and Lparse 45

List of Figures

2.1 MXG General Solving Scheme 10

3.1 MXG Problem Specification for Graph Colouring 17

5.1 $R_{x,y}$: Answer to $P(x,y)$ w.r.t. structure $\mathfrak{A}$ 32
5.2 $S_{x,z}$: Answer to $Q(x,z)$ w.r.t. structure $\mathfrak{A}$ 33
5.3 $T_{x,y,z}$: Answer to formula $P(x,y) \wedge Q(x,z)$ for Example 1 34
5.4 $W_{x,y,z}$: Answer to formula $P(x,y) \vee Q(x,z)$ for Example 1 34
5.5 $V_{x,y}$: Answer to formula $\exists\ z\ : P(x,y) \wedge Q(x,z)$ w.r.t. structure $\mathfrak{A}$ 35
5.6 $T^{z}_{x,y}$: Answer to formula $P(x,y) \wedge Q(x,z)$ for Example 1 39
5.7 $W^{z}_{x,y}$: Answer to formula $P(x,y) \vee Q(x,z)$ 40

6.1 Performance on Graph K-Colouring 49
6.2 Performance on Latin Square 50
6.3 Performance of MXG on Latin Square With/Without Cardinality Constraints 51
6.4 Performance on Blocked Queens 52
6.5 Performance on Social Golfer 53
6.6 Performance on Bounded Spanning Tree 54

Chapter 1

Introduction

A program is declarative if it describes what its output is like, rather than how to produce it. A declarative programming language for search problems provides users with a syntax with which to describe the relationship between instances of their problem and solutions. A "solver" for the language takes such a specification, together with an instance, and produces a solution for the instance if there is one. The use of such tools can greatly reduce the effort required to obtain effective practical solutions to a wide variety of problems, which otherwise would require a significant investment in development of problem-specific algorithms and implementations.

Descriptive complexity theory [28] seeks to characterize complexity classes by the type of logic needed to express the languages in them. Results from descriptive complexity allow viewing logics as "declarative programming languages" for problems in corresponding complexity classes. This is an idea that is well known, but has been widely considered of largely theoretical interest. Claims that it is not of practical interest can be roughly categorized as based on two objections:

- **The Language Objection** Logic is unsuitable as a practical modeling language: users with no formal logic background will not adopt the syntax, and in any case it does not provide the features necessary to support effective modeling of real problems.
- **The Solver Objection** Practically effective solvers cannot be constructed for such "general" or "abstract" languages as unrestricted classical logics.

Fagin's theorem [17], a seminal result in descriptive complexity, states that the classes of finite structures definable in existential second order logic ($\exists SO$), are exactly those in the complexity class NP. The theorem suggests a natural declarative problem solving approach for NP-complete problems: Represent the problem with an $\exists SO$ formula ϕ, and solve instances by (uniform) reduction to SAT or some other fixed NP-complete problem. When we consider NP-search problems, a solution is a witness to the existentially quantified relation variables of an $\exists SO$ sentence. The task becomes that of expanding a given structure to give suitable interpretations for those relation symbols. For ϕ a formula in logic $\mathcal{L}$, the task is *$\mathcal{L}$-model expansion* (abbreviated as $\mathcal{L}$-MX). When the logic $\mathcal{L}$ is classical first order logic (FO), we have FO-MX. The decision problem for FO-MX (does a suitable expansion exist), is the same as the model checking problem for ∃SO, so Fagin's theorem tells us that FO-MX can describe exactly the NP search problems (i.e., search problems whose associated decision problem is in NP).

In [44], Mitchell and Ternovska proposed to take the task of Model Expansion as the formal basis for a general framework for declarative programming for search problems. Descriptive complexity results establish the basis for applying the framework for declarative programming of search and decision problems from various complexity classes, based on the choice of logic. They specifically proposed to based a system for solving NP-search problems on FO(ID), an extension of FO with inductive definitions. Formally, for FO(ID)-MX can represent the same problems as FO-MX, but the use of inductive definitions can make a very significant difference in the ability to practically model problems. FO(ID) addresses some challenges related to the 'language objection': It is based on classical logic, which has a simple semantics for the user to understand, but the inductive definitions are convenient for defining some problems which are very difficult to define in pure FO-MX.

Cook's theorem [11], that every NP problem can be reduced in polynomial time to SAT, together with the impressive solving performance of SAT solvers, suggests that the following solving method could be practical: Given an instance, and the FO(ID)-MX specification of the problem, reduce the question of existence of a solution for the instance to to a SAT problem, and then run a SAT solver. This reduction is called *propositionalisation*, or *grounding*.

Polynomial-time grounding for FO-MX can be obtained naively by simple substitution, and replacing each universal (existential) quantifier with a (possibly large) conjunction (disjunction). Most grounding systems we are aware of use some version of this simple approach. Grounding for MX can be seen as a generalization of database query answering, and a grounding method based on a generalized relational algebra was defined and used in

[49] to obtain a theoretical result about grounding for FO-MX for a special class of FO formulas. Here, we extend and adapt that approach to produce a practical grounding scheme for general FO-MX. Using the relational join in a grounding algorithm has been done in practice [31], but only for constructing a guard for possible variable substitutions, and in a very restricted syntax. The main module of most grounders that we are aware of generate ground rules by simply instantiating variables [53, 31, 22, 30, 43]

Grounding for FO(ID) logic generates formulas of propositional calculus with inductive definitions (PC(ID)). Solving PC(ID) formulas requires either to constructing an effective solver for PC(ID) or producing a reduction of PC(ID) SAT that results in overall effective performance. Neither of these tasks is straightforward (but see [39, 41]) and [50] for work on the problems), and currently it is not clear what the most effective way to solve such formulas is. Here, to demonstrate the feasibility of our overall approach and methods without solving this problem, we restrict our implementation to classes of inductive definitions, as described in Section 2.4.3, which can be effectively handled by application of classical techniques.

We demonstrate, by producing an implemented system, called MXG, that FO(ID)-MX and grounding can be used to produce solver technology that is competitive, in terms of performance and convenience, with some well-established competing technology.
To have a more feasible modeling language, we add Cardinality Constraints to our language. MXG version 0.172 is the current solver of FO(ID+Card)-MX framework, which can handle a restricted form of cardinality constraints, and a fragment of inductive definitions by using a SAT solver. In more detail, we:

- add division, and a general union to the extended relational algebra of [49] and prove soundness and completeness of a grounding algorithm for FO-MX based on this.

- define the notion of *Extended-Hidden relations*, a more space efficient version of *Extended relations*, and generalize relational algebra operations to them.

- add a restricted semantics of *Cardinality Constraints* to FO(ID) logic, to provide a convenient way for user to express cardinality properties. We denote FO(ID) with cardinality constraints by FO(ID+Card). A (SAT+Card) solver should be used for solving, FO(ID+Card) formulas.

- present a concrete solver language for FO(ID)-MX, which extends classical FO with several features: sorts, order, inductive definitions, cardinality constraints.

- produce a grounder for FO(ID+Card)-MX, based on applying relational algebra operations on extended-hidden relations.

We implemented MXG0.172, based on this language and the grounding technique. (In this thesis, when we say MXG, we mean MXG0.172). To avoid dealing with complexity of PC(ID), MXG can handle only a fragment of inductive definition. We present sample specification for a number of benchmark problems in MXG, and compare the performance of MXG and our specifications with other systems, both in terms of efficiency of grounding and effectiveness as a solver. Our performance results show MXG to be competitive with high-quality ASP systems, smodels [54], DLV [33], clasp [20], and another FO(ID)-MX system, MidL (developed independently and concurrently with ours) [41], sometimes being less effective but often being more effective. Our work provides support for several claims:

- FO(ID)-MX is feasible as the basis for practical technology for solving NP-hard search problems.
- Grounding based on relational algebra is practical.
- Using languages with rich syntax is feasible, whereas some researchers have argued that very restrictive syntax is necessary for practical effectiveness of logic-based systems.
- Inductive definitions and cardinality constraints can be added to FO-MX and used effectively (even in rather limited forms).
- Solving by grounding to SAT works even with some form of inductive definitions and cardinality constraints .
- Having a SAT+Card ground solver is useful in practice.

The outline of the thesis is as follows. Chapter 2 we present the mathematical background of MXG, and justify our choice of language and solving method. Chapter 3 gives the syntax of language for MXG. In Chapter 4 we explain the solving method of MXG. In Chapter 5 we describe the grounding algorithm used in MXG. In Chapter 6 we compare the performance of MXG with ASP tools and MidL, on some benchmark problems. Chapter 7 presents a set of other declarative programming framework that are most closely related to MXG. Chapter 8 has the conclusion and future work.

Chapter 2

MXG Overview

The task of Model Expansion (MX) was proposed in [44] as the logical basis for a declarative constraint programming framework for solving search and decision problems. In this chapter we will explain the mathematical background for MX framework, and explain the choice of the logic and solving approach on which our MX solver MXG is based.

2.1 MX Mathematical Overview

Before we give the formal definition of model expansion task, we provide some necessary background from mathematical logic.

A *vocabulary* (or language) σ is a collection of constant symbols $(c_1, \ldots, c_n)$, relation or predicate symbols $(P_1, \ldots, P_m)$ and function symbols $(f_1, \ldots, f_l)$. Each relation and function symbol has an associated arity. A *σ-structure* $\mathfrak{A} = (A; \{c_i^{\mathfrak{A}}\}, \{P_i^{\mathfrak{A}}\}, \{f_i^{\mathfrak{A}}\})$ consists of a universe A together with an interpretation of:

- each constant symbol c_i from σ as an element $c_i^{\mathfrak{A}} \in A$;
- each k-ary relation symbol p_i from σ as a k-ary relation on A; that is a set $P_i^{\mathfrak{A}} \subseteq A^k$;
- each k-ary function symbol f_i from σ as a function $f_i^{\mathfrak{A}} : A^k \rightarrow A$.

A structure $\mathfrak{A}$ is called finite if its universe A is a finite set.

For example if σ has constant symbol 0, a binary relation symbol $<$, and one binary function symbol $+$, then one possible finite structure for σ is $\mathfrak{A} = (A; 0^{\mathfrak{A}}, <^{\mathfrak{A}}, +^{\mathfrak{A}})$, with universe of discourse $A = \{0, \ldots, 1000\}$ where $0^{\mathfrak{A}} = 0$, $<^{\mathfrak{A}}$, and $+^{\mathfrak{A}}$ have respectively the interpretation of less than, and sum modulo 1000.

A logic $\mathcal{L}$ is defined by a set of formulas of $\mathcal{L}$, together with a satisfaction relation which says when a formula ϕ of $\mathcal{L}$ is true in a structure $\mathfrak{A}$. Here we restrict our attention to finite structures, and to extensions of classical first order logic (FO), because finite model expansion for FO is the logical task underlying MXG.

A *σ-formula* ϕ of FO is constructed in the standard way with atomic formulas over vocabulary symbols σ and an infinite set of variables, and the usual inductive closure under connectives $\neg, \wedge, \vee$, and quantifiers $\exists, \forall$. If all variables of a formula are quantified (bound), the formula is called a *sentence*. A variable is *free* in ϕ, if it is not quantified in formula ϕ. *vocab*(ϕ) is the collection of exactly those function and relation symbols which occur in ϕ.

A σ-sentence ϕ of FO expresses a proposition in the language σ, which is either true or false when interpreted by a σ-structure $\mathfrak{A}$. If ϕ is true in structure $\mathfrak{A}$, we say that $\mathfrak{A}$ *satisfies* ϕ, or $\mathfrak{A}$ is a model for ϕ, written $\mathfrak{A} \models \phi$. If a formula ϕ has free variables, they must be interpreted as specific elements in the universe A before ϕ gets a truth value under structure $\mathfrak{A}$. An *object assignment* ω for a structure $\mathfrak{A}$ is a function from variables to the universe A. We write $\mathfrak{A} \models \phi[\omega]$ to indicate that structure $\mathfrak{A}$ satisfies formula ϕ when the free variables of ϕ are interpreted according to object assignment ω.

The satisfaction relation $\models$ for FO is defined recursively on the structure of a formula:

- $\mathfrak{A} \models P(t_1, \ldots, t_n)[\omega]$ iff $(t_1^{\mathfrak{A}}[\omega], \ldots, t_n^{\mathfrak{A}}[\omega]) \in P^{\mathfrak{A}}$.
- $\mathfrak{A} \models \neg C[\omega], \mathfrak{A} \models (D \wedge E)[\omega], \mathfrak{A} \models (F \vee G)[\omega]$ iff $\mathfrak{A} \not\models C[\omega]$, $\mathfrak{A} \models D[\omega]$ and $\mathfrak{A} \models E[\omega]$, $\mathfrak{A} \models F[\omega]$ or $\mathfrak{A} \models G[\omega]$ respectively.
- $\mathfrak{A} \models (\forall x C)[\omega]$ iff $\mathfrak{A} \models C[\omega(a/x)]$ for all $a \in A$. [1]
- $\mathfrak{A} \models (\exists x C)[\omega]$ iff $\mathfrak{A} \models C[\omega(a/x)]$ for some $a \in A$.

A *theory* is a set of sentences. A σ-structure $\mathfrak{A}$ is a model of a theory T in FO, written $\mathfrak{A} \models T$, iff for every sentence ϕ of T, $\mathfrak{A} \models \phi$. *vocab*($T$) is the union of *vocab*(ϕ) for all $\phi \in T$.

- A set of FO formulas, Φ, is *Satisfiable* iff it has a model. That is for some structure $\mathfrak{A}$, and some object assignment ω, $\mathfrak{A} \models \phi[\omega]$ for every $\phi \in \Phi$.

[1]If x is a variable and $a \in A$, then the object assignment $\omega(a/x)$ is the same as ω except $\omega(x) = a$.

- The *Model Checking* problem is to decide if a given structure $\mathfrak{A}$, satisfies a given theory T: $\mathfrak{A} \models T[\omega]$.

- The *Model Expansion* problem is : Given a theory T with vocabulary $vocab(T) = \sigma \cup \varepsilon$, and a σ-structure $\mathfrak{A} = (A; \sigma^{\mathfrak{A}})$, is there a $\sigma \cup \varepsilon$-structure $\mathfrak{B} = (A; \sigma^{\mathfrak{A}}, \varepsilon^{\mathfrak{B}})$ which is an expansion of $\mathfrak{A}$ to ε, such that $\mathfrak{B} \models T$.

We call σ the 'instance vocabulary', ε the 'expansion vocabulary', $\mathfrak{A}$ the 'instance structure', $\mathfrak{B}$ the 'expansion structure'.

The model expansion problem is between model checking and satisfiability problems, where part of the desired model is given and we ask for an expansion that satisfies the given theory. For the cases $\varepsilon = \emptyset$ and $\sigma = \emptyset$ we have, respectively, model checking and satisfiability, not model expansion. The three problems can be defined for any logic $\mathcal{L}$, and we will refer to the model expansion problem for logic $\mathcal{L}$ as $\mathcal{L}$-MX.

In the MX framework, the idea is to cast search problems as the logical task of model expansion. This is a natural way to model a search problem: specify the relationship between an instance and its solution by a set of sentences of some suitable logic $\mathcal{L}$ (theory T), and describe the instance by a finite instance structure ($\mathfrak{A}$). A solution for an instance $\mathfrak{A}$ is given by an interpretation of the vocabulary symbols of the formula that are left un-interpreted by $\mathfrak{A}$.

Example. The graph 3-colouring problem can be axiomatized as a FO-MX problem. The input structure is a graph $G = (Vtx; Edge)$, and theory T is the collection of following formulas over the vocabulary $vocab(T) = \{Edge, R, B, G\}$:

$\forall$ v : R(v) $\vee$ B(v) $\vee$ G(v);
$\forall$ v : $\neg$ R(v) $\vee\neg$ B(v);
$\forall$ v : $\neg$ R(v) $\vee\neg$ G(v);
$\forall$ v : $\neg$ B(v) $\vee\neg$ G(v);
$\forall$ u v : Edge(u,v) $\supset \neg$ (R(v) $\wedge$ R(u));
$\forall$ u v : Edge(u,v) $\supset \neg$ (B(v) $\wedge$ B(u));
$\forall$ u v : Edge(u,v) $\supset \neg$ (G(v) $\wedge$ G(u)).

The instance vocabulary is $\sigma = \{Edge\}$, and expansion vocabulary is $\varepsilon = \{R, B, G\}$. A structure $\mathfrak{B}$ which is an expansion of G to $vocab(T)$ is a model for T, $\mathfrak{B} \models T$, iff $R^{\mathfrak{B}}, B^{\mathfrak{B}}, G^{\mathfrak{B}}$ comprise a proper 3-colouring of vertices in G.

2.2 MXG: A FO-MX Solver

For an implementation of the MX framework, we must choose a logic which will be the underlying logic for the language and design a modelling language for users. The choice of logic underlying the modelling language depends in part on the complexity class of problems you want to model and solve. The study of the relationship between logical definability and computational complexity is called descriptive complexity [28]. Results show that, in certain cases, a logic $\mathcal{L}$ *captures* a certain complexity class $\mathcal{C}$, which means that any problem expressed in $\mathcal{L}$ is in complexity class $\mathcal{C}$, and moreover all problems in complexity class $\mathcal{C}$ can be expressed in logic $\mathcal{L}$. Thus for modelling problems in a certain complexity class, the we may choose as our underling logic one that captures the desired class. This allows us to fine-tune our language for desired complexity classes and get a universal framework for representing problems in that class, without imposing ad-hoc restrictions on syntax. This is an important issue for us, to found all of our choices for implementing MXG on sound theories.

Fagin's theorem [17], one of the first results in descriptive complexity, states that existential second order logic, $\exists SO$, captures NP. Based on this correspondence, first order logic was proposed as the logic underlying the MXG language. For a fixed FO theory T in FO-MX, expansion vocabulary symbols behave as existentially quantified second order variables. Thus FO-MX has the same power as ($\exists$SO) over finite structures and can express any problem in NP.

Note that this result is for the case that theory T is fixed, and only instance structure $\mathfrak{A}$ can change and be specified by input. This is the 'Parameterised MX' setting in contrast to 'Combined MX' setting [56] where both instance structure $\mathfrak{A}$ and theory T are part of the instance. FO-MX in the combined setting is NEXPTIME-complete (See [44].)

In the 'Parameterised' setting, there is a formal distinction between theory T and instance finite structure $\mathfrak{A}$, which suggests passing problem specification and instance description separately to MXG. This separation in practice is quite useful, as for example in many applications the problem specification will be carefully refined then used for many instances. They can be reasoned about separately and specific pre-processing methods be applied to each of them. We may use different languages for describing problem specification and instance in MXG, as they are really different sorts of objects.

2.3 MXG Solving Approach

There are many possibilities for building a solver for the MX framework. For example one could directly implement a search algorithm for finding the desired expansion structures, or use a theorem prover to check the validity of problem for different expansion structures. Cook's theorem [11], that every problem in the complexity class NP can be reduced in polynomial time to SAT, suggested an alternative scheme: (1) For each problem P, implement a reduction to SAT; (2) Given an instance of P, apply the reduction to produce a propositional formula, then run the best SAT solver available to find a satisfying assignment (thus solution) if there is one.

SAT-based techniques have been shown to be very effective in solving a number of NP-hard problems, and in some cases have proven more effective than special-purpose programs of the day in different areas, such as planning [24] [29], model checking in hardware verification[7] and especially Bounded Model Checking (BMC). Implementations of other logic-based tools routinely either use a SAT solver as the core engine, or implement variants of SAT solver methods. An attraction of basing systems on SAT solvers is that performance of standard SAT solvers improves regularly, and one can generally 'plug in' the latest and best. SAT solvers are currently effective in a number of domains, and recent work suggests much more potential.

Selecting SAT-based solving approach in our framework requires performing a generic reduction from FO-MX axioms to SAT. This reduction is called "grounding" or "instantiation": it substitutes variables of each axiom with all the possible values for them, and evaluates out the instance ground (instantiated) atoms with their values from the instance structure. This leaves us with just expansion atoms in ground axioms. We call such an axiom a *reduced ground axiom.* Then each reduced ground axiom is converted to a propositional formula by mapping each ground atom to a unique propositional variable. The set of propositional formulas are then transformed to CNF by a polynomial-time procedure. A SAT-solver then searches for satisfying assignments of the CNF formula. The true literals of a satisfying assignment are "un-ground" to FO ground atoms, using the mapping created in grounding phase. Interpretation of expanded vocabulary symbols is specified by the "un-grounded" literals. The grounding algorithm should be sound and complete, which means it associates each first order logic solution to exactly one propositional solution, and no solution is lost or created incorrectly. We explain the grounding algorithm implemented

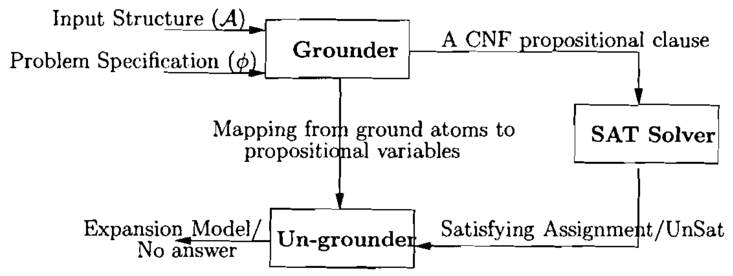
Input Structure ($\mathcal{A}$)
Problem Specification (ϕ)
Grounder
A CNF propositional clause
SAT Solver
Mapping from ground atoms to propositional variables
Un-grounder
Satisfying Assignment/UnSat
Expansion Model/ No answer

Figure 2.1: MXG General Solving Scheme

in MXG, Gnd-Hidden, in Chapter 5, and prove its soundness and completeness.

The general solving scheme for MXG is shown in Figure 2.1. The problem description and instance structure are given in separate files to MXG. The module identified as "Grounder" in Figure 2.1 generates the propositional clauses in DIMACS format and sends them to a SAT solver. The mapping from FO ground atoms to propositional variable is sent to "un-grounder". If the SAT solver finds a satisfying, it assignment is translated back to a FO model according to the mapping by "un-grounder" module.

2.4 Extensions to First Order Logic

Although any NP problem is expressible in FO-MX, it is useful to consider a language based on certain extensions of FO which provide some facilities to improve convenience of modelling. In particular, we extend standard first order logic with multiple sorts, ordered domains, inductive definitions [15] [13] [14] and cardinality constraints. We denote the problem of MX with this logic by FO(ID+Card)-MX. Note that adding these features to FO does not give any extra expressiveness power: FO(ID+Card)-MX has the same power and complexity as FO-MX. These features just come in very handy for expressing certain properties.

2.4.1 Multi-sorted Logic

Defining and using types for vocabulary symbols helps rejecting some erroneous or undesirable specifications. It also makes axiomatization more understandable. In Multi-Sorted FO the universe is partitioned into a set of sorts, and we specify for each variable which sort it is to range over. We can reduce the multi-sorted case to single-sorted by using *domain predicates*, introduced for each type. For a sentence $\phi = \forall x : P(x)$, where P is declared over type D, in single-sorted FO the domain predicate $D(x)$ is introduced, and added to ϕ to assure x ranges over values of D: $\forall x : (D(x) \supset P(x))$. For $\phi = \exists x : P(x)$ domain predicate is conjoined with the quantified sub-formula: $\exists x : (D(x) \wedge P(x))$. In these formulas, we call D, guard for x.

2.4.2 Order

We take all structures to be ordered. A total ordering on elements of each sort is defined by the ordering elements are specified in the instance structure. The imposed ordering allows us to extend the instance vocabulary with binary relation symbols $<, >, \neq, =, \leq, \geq$ over each sort, with their expected meaning regarding the ordering of elements of each sort. Constants MIN and MAX denoting the first and last elements of each sort, and the binary relation SUCC with its natural semantics for each sort are also provided.

2.4.3 Inductive Definitions

ID-logic [15] [13] [14] extends classical logic with inductive definitions. Both monotone and non-monotone induction are formalized in a natural way in the ID-logic. The classical part of the logic has the usual classical semantics. The semantic of inductive definitions is based on the two-valued well-founded semantics of logic programs.

FO has no feature for expressing recursive properties such as reachability. These properties are conveniently expressible with inductive definitions. To remedy this, we use FO(ID), a fragment of ID-logic with FO as the classical logic, for MXG. FO(ID) is more expressive than FO, but FO(ID)-MX is not more expressible than FO-MX, as we have the power of $\exists SO$ logic in FO-MX. Recursive properties can be expressed in FO-MX, but axiomatizations of this sort are often not intuitive, and may be very hard to produce.

The general semantics is complex, and it is not clear how to construct an effective solver for FO(ID)-MX in general. Two possible approaches are to:

- produce a reduction of inductive definitions to SAT that captures the semantics of non-monotone inductive definitions(See [50]);
- reduce to an extension of propositional logic with inductive definitions and build a ground solver for this language (See [39] [41]).

Neither reduction to SAT, nor direct implementation of a solver is trivial. In this stage of MXG's development, we restrict our implementation to handle two special fragments of inductive definitions of FO, *Horn-ID* and *Comp-ID*, which we can handle simply and effectively using standard techniques. We call an inductive definition "well-behaved", if it is a Horn-ID, or a Comp-ID. Although MXG can not compute general inductive definitions at this time, our "well-behaved" inductive definitions are useful for many problems.

Horn-ID:

An inductive definition is a Horn-ID, if the defined predicate occurs only positively in the definitions, and all predicate symbols in its body are instance predicates or are effectively instance predicates (as they have already been computed during grounding). Such a definition is the same as a 'definite' logic program, for which a unique minimum model exists, that is a model with with the minimum number of true atoms. If we treat $\leftarrow$ as $\supset$, then we get a set of Horn formulas (clause with only one positive literal). This set has a unique minimum model, which is the same as the same the model of the inductive definition. The minimum model for Horn theory is found by a polynomial time procedure [27] [16]. MXG has a built-in module that precomputes this model. The defined predicate is then treated as an instance predicate for the remainder of the grounding of this specification.

Example: To find the distance of vertices in a graph $G = (V; Edge)$ from a particular vertex $Start \in V$ we can use the following inductive definition, where $Dist(v, n)$ is true iff the distance of $v, Start$ is n :

$$\{Dist(v,n) \leftarrow (v = Start) \wedge (n = MIN)$$
$$Dist(v,n) \leftarrow Dist(u,m) \wedge Edge(u,v) \wedge SUCC(m,n)\}$$

Horn-ID definitions can be used for the preprocessing purposes. Instead of writing a program to construct a new relation over elements of input structure, one might define the new relation by a Horn-ID, and use MXG to compute the interpretation of relation. For example, if distances of vertices from a fixed vertex are required in a graph problem, it can be computed either by writing an individual program to read the instance graph and compute distances, or easily by adding the above definition to problem axioms in MXG. Of course only polynomial time preprocessing can be computed by a Horn-ID definition.

Comp-ID

We will say an inductive definition is a *Comp-ID* iff it is defined over a well-founded order. If a definition is not a Horn-ID, MXG replaces it with its completion [10], and compute the model of completion. This replacement is only correct for Comp-IDs. User should be aware that wrong solutions may be found if a non-Comp-ID is defined.

Example : In a structure with the universe a prefix of *Natural* numbers, $S = [0 \dots \nu]$, we can define predicates *Odd*, *Even* with their natural meanings by two definitions:

{ Even(n) ← n=MIN // MIN is 0 in sort S
 Even(n) ← ¬ Odd(n)
 Even(n) ← Odd(n') ∧ SUCC(n',n)}
{ Odd(n) ← Even(n') ∧ SUCC(n',n)}

Neither of these definitions is a Horn-ID: as they have rule bodies containing expansion predicates other than head predicate. On the other hand, they are defined over the well-founded order of the Natural numbers. So they are Comp-ID's and equivalent to their completion.

Convenience of General Inductive Definitions

Having inductive definitions fully supported provides a very convenient paradigm for expressing recursive properties. For example in the *Hamiltonian Cycle* problem for an undirected instance graph $G = (Vtx; Edge)$, the reachability of vertices from a fixed vertex MIN, *Reached*, through Hamiltonian cycle edges, HC, can can be defined as:

{ Reached(v) ← v=MIN
 Reached(v) ← Reached(v') ∧ HamCycle(v',v) }

However MXG can handle only a well-behaved fragment of inductive definitions. The inductive definition for *Reached* is not well-behaved. To axiomatize Hamiltonian Cycle problem correctly in MXG, we need to introduce an auxiliary binary predicate symbol. For example we may introduce *Map*(Vtx, Vtx) and express the problem by the following first order sentence:

//Map is a bijection from natural numbers[1..vertices#] to vertices
∀ v : ∃ n : Map(n,v)
∀ n : ∃ v : Map(n,v)
∀ v n1 n2: ((Map(n1,v) ∧ Map(n2, v)) ⊃ n1=n2)

```
∀ n v1 v2: ((Map(n,v1) ∧ Map(n, v2)) ⊃ v1=v2)
//if vertices v1, v2 are mapped to 2 consecutive natural numbers,
// there should be an edge between them
∀ n1 n2 v1 : ((Map(n1, v1) ∧ (SUCC(n1, n2) ∨ (n1=MAX ∧ n2 = MIN))) ⊃
(∃ v2 : (Map(n2, v2) ∧ (Edge(v1, v2) ∨ Edge(v2, v1)))))
//HC(u,v) is defined based on the vertices mapped to consecutively to [1..vertices#]
∀ v u : (HC(v, u) ⇔ ((Edge(v,u) ∨ Edge(u,v)) &
(∃ n1 n2 : (Map(n1, v) ∧ Map(n2, u) ∧ (SUCC(n1, n2) ∨ (n1 = MAX ∧ n2 =MIN))))))
```

Auxiliary predicate *Map* defines a one-to-one mapping from natural numbers [1..vertices#] to vertices, and implies that every vertex appears exactly once as the 'to-vertex' in edges of hamiltonian cycle. This example illustrates that indeed recursive properties can be expressed in FO-MX, but it is more convenient to state them with inductive definitions. One of our goals in the future implementations is to handle inductive definitions broadly, with no restriction.

2.4.4 Cardinality Constraints

Expressing counting properties is not simple in FO-MX, and usually requires introducing auxiliary relations to count elements of a set. MXG together with a SAT+Card solver provides some simple cardinality options. Cardinality constraints are reduced to propositional cardinality constraint clauses. These clauses are not supported by standard SAT solvers, and a SAT+Card solver is needed for solving them. MXG uses a SAT+Card solver, MXC (available from [4]). MXC is not the only SAT+Card solver, but is currently the only one that can be used for handling cardinality clauses generated by MXG, as there is no standard format for propositional cardinality clauses, and MXG generates clauses based on MXC's format. (If a standard format for cardinality constraints is established, it is easy to revise MXG to generate standard clauses.)

A cardinality constraint in MXC is of the form $\forall \overline{x} : \odot(\nu; \overline{y}; \phi(\overline{x}, \overline{y}))$. Cardinality symbols $UB, LB, CARD$ allow user to express respectively an upper bound, a lower bound and an exact bound on size of a set. The semantics of a cardinality constraint formula $\forall \overline{x} : \odot(\nu; \overline{y}; \phi(\overline{x}, \overline{y}))$ in MXG, where $\odot$ is one of the $UB, LB, CARD$ is that, for any choice of $\overline{x}$, size of set $\{\overline{y} : \phi(\overline{x}, \overline{y})\}$ is less than, greater than, or equal to ν respectively for $UB, LB, CARD$.

For example, axiom $\forall u : CARD(1; v; Edge(v, u))$ in a graph problem states that each vertex should have in-degree 1. (For precise syntax of cardinality constraints see Section 3.2.3). In FO-MX, one might express the above property by the FO sentence $\forall u : \exists v : (Edge(u, v) \wedge \forall w : (Edge(w, u) \supset w = v))$.

Presence of cardinality constraints does not change the expressiveness of language, and just facilitates stating counting properties. The *Social Golfer* [51] problem is a good example for demonstrating the facility provided by cardinality constraints. The problem is to try to schedule $Players = Groupsize * Group$ golfers into $Group$ groups of $Groupsize$ players over $Week$ weeks, such that no golfer plays in the same group with any other golfer more than once. Let $Groupsize$ be an instance constant, and $Plays(Players, Weeks, Groups)$ be an expansion predicate specifying a schedule. In FO(ID+Card)-MX the axiom $\forall\ w\ \ g : CARD(Groupsize; p; Plays(p, w, g))$ states that exactly $Groupsize$ players should play in each group, in each week.

In FO-MX, we need to declare auxiliary predicate $M(Weeks, Players, Groupsize)$ to assign a player# from $[1 \ldots Groupsize]$ to each player, for each week. To force exactly $Groupsize$ players be playing in each group, we state that players of the same group should be assigned distinct player#.

∀ w p: ∃ l: M(w,p,l)

∀ w g p1 p2 l : ((Plays(p1,w,g) ∧ Plays(p2, w, g) ∧ M(w,p2,l) ∧ M(w,p1,l)) ⊃ (p1 = p2))

At the time of developing MXG, only a restricted format of cardinality constraints were supported by MXC. The restricted syntax of cardinality constraints in MXG0.172 is due to the restriction of its SAT+Card solver, MXC at that time. Currently MXC can handle a more general cardinality clauses. We can adopt MXG to handle a more general cardinality constraints in next version, so that cardinality part $(\nu; \bar{y}; \phi(\bar{x}, \bar{y}))$ can appear in any FO formula, not only in the universal formulas.

Chapter 3

MXG Input Language

In this chapter we specify and explain the languages for problem specification and instance description, which are distinct, in MXG. Grammars for these languages is given in Appendices A and B.

The languages are in part a co-operative effort involving ourselves, the developers of another solver for FO(ID) model expansion, MidL [39] [41], at Katholieke Universitat Lueven (KU Leuven), and others. Differences between the languages of MXG and MidL primarily reflect differences in what is implemented in the respective systems. ***** explaining what is our contribution****** Specifically, the structure of problem specification file, ordered strctures, bounded quantifiers, cardinality constraints, built-in constants MIN, MAX and SUCC are proposed by us.
A problem specification file for MXG consists of 3 sections:

- **Given:** Has declarations of what is given with the instance, that is all types, and all instance vocabulary symbols.
- **Find:** Has declarations of the "solution vocabulary". The solution vocabulary is the collection of those expansion predicate and constant symbols that constitute a solution.
- **Satisfying:** has the set of axioms, plus declarations of any "auxiliary vocabulary" symbols. The axioms are arbitrary first order logic sentences, cardinality constraints, and inductive definitions. Auxiliary vocabulary is the collection of expansion predicate and constant symbols that are not really part of a solution, but are needed or useful

```
Given:   type Vtx Clr;
         Edge(Vtx, Vtx)

Find:    Colour(Vtx, Clr)

Satisfying:
         ∀ x y z : ((Colour(x, y) ∧ Colour(x, z)) ⊃ (y=z))
         ∀ x y : (Edge(x,y) ⊃ (∀ z : ¬(Colour(x, z) & Colour(y,z))))
         ∀ x: ∃ y : Colour(x, y)
```

Figure 3.1: MXG Problem Specification for Graph Colouring

for axiomatization.

We present the syntax of an MXG instance specification file, in Section 3.1. Then each of Sections 3.2.1, 3.2.2, and 3.2.3 describe in detail each the of sections Given, Find, and Satisfying.

In Sections 3.1 and 3.2 we give the syntax for the specification file, using the Graph Colouring problem as a running example to illustrate. The full specification for Graph k-Colouring problem is given in Figure 3.1. Types are Vtx (vertices) and Clr (colours); the instance vocabulary is the binary relation Edge; the solution vocabulary is the binary relation Colour. The axioms simply say that the interpretation of Colour must constitute a proper colouring of the given graph.

3.1 Instance Description Syntax

An instance description file specifies the elements for each type, and the interpretation of each instance vocabulary symbol. A type is specified by $SortName = [elemets]$, where *elements* is a range of natural numbers or characters, $i..j$, or a ';'-separated enumeration of natural numbers, characters or strings. A string element is constructed with $[A-Za-z0-9\_]^*$, starting with a capital letter.

The interpretation of each instance predicate is given by the set of its tuples, separated by ';'. Attributes of each tuple are separated by ','.

Example: If our instance is a graph, we may have a type Vtx which gives the set of vertices and a binary relation, Edge, giving the edges of the graph. For type Vtx,

Vtx = [1..4] in the instance file specifies elements of Vtx to be [1..4].

For problem predicate Edge:

Edge = {1, 2; 1, 4; 2, 3; 3, 4} in the instance file specifies tuples of relation Edge to be {(1,2), (1,4), (2,3), (3,4)}.

Comments come within /* */, or on a line after //.

3.2 Problem Specification Syntax

In the MXG problem specification file, each vocabulary symbol must be declared prior to its use. Instance vocabulary, solution vocabulary, and auxiliary vocabulary are respectively declared in Given, Find, and Satisfying sections. This separation makes the problem specification file more informative. By looking at the problem specification, one knows that the interpretation of symbols declared in Given part is given in instance description file, and a solution is an interpretation of symbols declared in Find part.

A predicate or constant symbol, as well as a type name, is a string of $[A-Za-z0-9\_]^*$ starting with an upper case letter $[A-Z]$. Variables are strings starting with a lower case letter. MXG does not have variable declarations. We explain how type of a variable occurrence in an axiom is determined in Section 3.2.3. Functions are not supported in MXG. Comments come within /* */, or on a line after //.

3.2.1 Given Section

Terms in MXG axioms are variables or constant symbols. Every term has a certain type, which must be one of the types declared in the Given section, by the keyword type. For graph the colouring example,

type Vtx Clr;

declares two types Vtx and Clr.

MXG treats the elements of each type as a prefix of the Natural numbers, by mapping elements of a type to Natural numbers based on the ordering they appear in instance file. For example type Clr defined with four elements by Clr=[Blue; Red; Yellow; Green] is mapped to $\{0, 1, 2, 3\}$, with Blue mapped to 0, Red mapped to 1, and so on. By this correspondence to Natural numbers, a total ordering on the elements of each type is imposed: Blue < Red < Yellow < Green. Ordering relations $<, >, =, \neq, \leq, \geq$ can be used over each type. Interpretation of ordering relations is induced by the mapping of type elements to natural numbers. Binary relation SUCC, with its natural semantics, for each type, and constants MIN and MAX denoting the first and last elements of each type, are also provided.

Instance vocabulary symbols, whose interpretations are given in the instance file, must be declared in Given part. An instance predicate is declared by specifying the type of each argument, in the following manner:

Edge(Vtx, Vtx)

declares Edge to be a binary predicate symbol, which must be interpreted by a relation on Vtx * Vtx.

An instance constant is declared by specifying its type:

ColRed : Col

which declares a constant ColRed of type Col.

3.2.2 Find Section

Solution vocabulary symbols are declared in the Find section, in the same form as declarations of instance vocabulary. For example,

Find : Colour(Vtx, Clr)

declares a binary solution predicate Colour over Vtx * Clr.

The interpretation of Colour is the solution of the colouring problem. MXG outputs an interpretation of the solution vocabulary symbols, if it finds a solution.

3.2.3 Satisfying Section

If, for axiomatization of a problem, vocabulary symbols other than instance and solution vocabulary are needed, they are declared in this section, and can be used in axioms following their declaration.

An axiom in an MXG Satisfying section is either a FO sentence, an inductive definition, or a cardinality constraint.

FO Sentences: A FO sentence may use all vocabulary symbols declared in the Given and Find parts, plus any auxiliary symbols declared before the sentence in the Satisfying part. They may also use the ordering relations $<, >, =, \neq, \geq, \leq$, binary predicate $SUCC$, constants MIN, and MAX, an infinite set of variables, connectives $\neg, \wedge, \vee, \supset, \Leftrightarrow$, quantifiers $\forall, \exists$ and parenthesis ().

Logical Symbol	$\forall$	$\exists$	$\wedge$	$\vee$	$\neg$	$\supset$	$\Leftrightarrow$
ASCII Representation	!	?	&	|	~	=>	<=>

Table 3.1: ASCII Equivalents for Logical Symbols

In MXG we declare types (sorts) of arguments to predicate symbols, not variables. Variables are not declared. The type of each variable, in a sentence, is inferred from its position in predicates, which must be consistent. Every variable in a sentence must appear at least once in a location that gives it a type. Formulas of the form, $\forall x \forall y < x : (y \neq x)$ are not acceptable in MXG, as x, y are not used in any typed predicate to determine their type.

Only terms of identical types can be compared by $<, >, =, \neq, \leq, \geq$, and the two arguments to SUCC must be of identical type. MIN and MAX are built-in constants denoting the first and last element of a type. Because arguments to predicate symbols are typed, their use is unambiguous. For example, in the sentence $\forall v : \exists c : Colour(v, c)$, the types of v and c are, respectively, Vtx and Clr. A first order formula $Q_1 x_1 : \ldots Q_1 x_n :$ can be abbreviated to $Q_1 x_1 \ldots x_n :$. Each quantifier Q binds all variables following the Q and preceding the next ':'.

Bounded quantifiers of the form $Q\alpha \odot \beta$ are supported, where Q is a quantifier, $\odot$ is a relational operator $(<, >, =, \neq, \geq, \leq)$, α, β are variables or constants and could be MIN or MAX. At least one of α or β must be a variable. Both sides of a comparison should be of the same type. In the graph colouring problem, the sentence $\forall v : \exists c > ColRed : Colour(v, c)$, for the given type Clr=[Blue; Red; Yellow, Green] and ColRed is interpreted as Red, states that each vertex v can be coloured by either Yellow or Green. Total ordering Blue < Red < Yellow < Green is inferred by the order of their appearances in type Clr.

The concrete syntax of an MXG file is obtained by replacing logical symbols with ASCII versions as given in Table 3.1

Inductive Definitions: An inductive definition is defined by a set of rules within $\{\}$. Each rule is of the form $Head \leftarrow Body$, where $Head$ is an atom whose predicate symbol is an expansion predicate, and is fixed in all rules of an inductive definition. $Body$ is a quantifier-free FO formula. Head (body) variables are implicitly universally (existentially) quantified.

For example to find the distance of vertices in a graph $G = (Vtx; Edge)$ from a particular vertex $Start \in Vtx$ we can use the following inductive definition:

$\{Dist(v,n) \leftarrow (v = Start) \wedge (n = MIN)$

$Dist(v,n) \leftarrow Dist(u,m) \wedge Edge(u,v) \wedge SUCC(m,n)\}$

$Dist(v,n)$ is true iff the directed distance from $Start$ to v is n.

As explained in Section 2.3, MXG can handle only some classes of "well-behaved" inductive definitions, and wrong solutions (i.e., structures which are not models) might be returned as solutions if a problem axiomatization has inductive definitions that are not "well-behaved". The language syntax of MXG allows writing inductive definitions that are not "well-behaved", and users should be careful to write only "well-behaved" inductive definitions.

Cardinality Constraints: cardinality constraints are provided for the convenience of expressing counting constraints. MXG handles cardinality constraints of the following form: $\forall \overline{x} : \odot(\mu; \overline{y}; \phi(\overline{x}, \overline{y}))$. Here, $\odot$ is one of *CARD, UB, LB* which stand for Cardinality, Upper-bound and Lower-bound, respectively. $\phi(\overline{x}, \overline{y})$ is a FO formula with free variables $\overline{x}, \overline{y}$. μ is either a natural number or an instance constant symbol which can be of any type. For each $\overline{a} : \overline{x} \to A$, the formula constrains the number of $\overline{b} : \overline{y} \to A$ for which $\phi(\overline{a}, \overline{b})$ is true in the expansion structure $\mathfrak{B}$:

$$\forall \overline{a} : lb \leq |\{\overline{b} : \overline{y} \to A, \mathfrak{B} \models \phi(\overline{a}, \overline{b})\}| \leq ub.$$

In the above, values of lb, ub, the lower bound and upper bound, are set accordingly for $UB, CARD$, and LB:

- $lb = 0, ub = BOUND$ for UB,
- $lb = ub = BOUND$ for $CARD$,
- and $lb = BOUND, ub = k$ for LB, where k is the size of set $\{\overline{b} : \overline{y} \to A\}$.

Value of $BOUND$ is:

- If μ is a natural number, $BOUND$ is the value of μ.
- If μ is a constant symbol, $c : D$, c denotes an element of type D. If that element is the ith element in the order for D, then $BOUND$ is i.

For example $\forall u : UB(1; v; Edge(v,u))$ requires that the in-degree of each vertex is at most one.

Chapter 4

MXG Solving Method

In this chapter we present in detail the "grounder" and "un-grounder" modules of MXG, as illustrated in Figure 2.1.

4.1 "Grounder" Module

Grounding for any high-level language is the process of eliminating variables, by replacing them with constant symbols, which represent elements of the appropriate type. In MXG, a new constant symbol, c_a, is introduced for each element $a \in A$ for instance structure $\mathfrak{A} = (A; \sigma^{\mathfrak{A}})$. We denote the collection of these new constant symbols introduced for universe A, by $\tilde{A}$. The instance σ-structure $\mathfrak{A}$ is expanded to a $(\sigma \cup \tilde{A})$-structure $\mathfrak{A}^* = (A, \sigma^{\mathfrak{A}}, \tilde{A}^{\mathfrak{A}^*})$, where $c_a^{\mathfrak{A}^*} = a$ for all $c_a \in \tilde{A}, a \in A$.

Throughout this thesis, $\mathfrak{A} = (A; \sigma^{\mathfrak{A}})$ is a finite instance structure, $\tilde{A}$ is the collection of constant symbols, $\mathfrak{A}^* = (A; \sigma^{\mathfrak{A}}, \tilde{A}^{\mathfrak{A}^*})$ is the instance structure expanded with constants $\tilde{A}$, σ is the instance vocabulary, and ε is the expansion vocabulary, unless otherwise stated.

Def.(Reduced Grounding): Let ϕ be a formula over vocabulary $\sigma \cup \varepsilon$. A *reduced grounding* of ϕ with respect to (w.r.t.) finite structure $\mathfrak{A} = (A, \sigma^{\mathfrak{A}})$, is a ground formula ψ over vocabulary $\varepsilon \cup \tilde{A}$ only, such that, for any structure $\mathfrak{B} = (A; \sigma^{\mathfrak{A}}; \tilde{A}^{\mathfrak{A}^*}; \epsilon^{\mathfrak{B}})$ and any object assignment ω, $\mathfrak{B}[\omega] \models \phi$ iff $\mathfrak{B} \models \psi$.

A reduced grounding exactly defines the set of solutions for the instance $\mathfrak{A}$. MXG obtains a reduced grounding of a formula, by grounding (instantiating) the formula and simultaneously 'evaluating out' the instance vocabulary.

We later in Chapter 5 explain a polynomial time grounding algorithm, Gnd-Hidden, that

MXG uses for computing the reduced grounding of a FO axiom w.r.t. instance structure $\mathfrak{A}$.

The "grounder" module of MXG generates the reduced grounding of each problem axiom w.r.t. instance structure $\mathfrak{A}$. The resulting ground FO formulas must be transformed to propositional CNF clauses before being passed to a SAT solver. In Section 4.1.1 and Section 4.1.2 we explain this transformation in MXG.

4.1.1 Converting a Reduced Ground FO Formula to Clausal Form

A FO clause is a disjunctive FO formula. Conversion of an arbitrary ground FO formula to clausal form is a straightforward procedure. Polynomial time transformation requires use of new variables, sometimes called *Tseitin* variables [1] [55]. Procedure Convert-To-FO-Clause(α, ψ) in MXG, is an implementation of this procedure. It takes as input a reduced ground FO formula ψ, and a new *Tseitin* predicate symbol α and recursively operates on the structure of ψ to generate a set of ground FO clauses ψ_C equivalent to ψ [2]. A Tseitin predicate symbol, is a 0−ary predicate symbol that can be interpreted either to true or false. Convert-To-FO-Clause associates each sub-formula χ_p of ψ, with a new Tseitin predicate symbol p. We denote the collection of these new predicate symbols introduced in this transformation by Γ. Convert-To-FO-Clause generates clauses to ensure that for any expansion structure $\mathfrak{B} = (A; \sigma^{\mathfrak{A}}; \varepsilon^{\mathfrak{B}}; \Gamma^{\mathfrak{B}})$, $p^{\mathfrak{B}} \Leftrightarrow \chi_p^{\mathfrak{B}}$. In the following, by "clauses for ϕ" we mean the natural set of clauses ϕ' over exactly the atoms of ϕ, such that ϕ and ϕ' are equivalent. For example, clauses that are generated for $\alpha \Leftrightarrow (\alpha_1 \vee \cdots \vee \alpha_n)$ are: $(\neg\alpha \vee \alpha_1 \vee \cdots \vee \alpha_n)$, and a set of binary clause : $(\alpha \vee \neg\alpha_1), \ldots, (\alpha \vee \neg\alpha_n)$

Procedure Convert-To-FO-Clause(α, ψ) :

1. If ψ is an atom, generate clauses for $\psi \Leftrightarrow \alpha$ and return.

2. If $\psi = \chi_1 \wedge \cdots \wedge \chi_n$, add Tseitin predicate symbols $\alpha_1, \ldots, \alpha_n$ to Γ. Generate clauses for equivalence $\alpha \Leftrightarrow (\alpha_1 \wedge \cdots \wedge \alpha_n)$. Call Convert-To-FO-Clause(α_i, χ_i) for each χ_i and α_i, $i \leq n$.

3. If $\psi = \chi_1 \vee \cdots \vee \chi_n$, add Tseitin predicate symbols $\alpha_1, \ldots, \alpha_n$ to Γ. Generate clauses

[1]The polynomial transformation of an arbitrary propositional formula to clausal form, was first introduced by Vladimir Tseitin, and the name of the extra variables needed in this transformation are named Tseitin variables after his name.

[2]Two FO theories T_1, T_2 are equivalent iff every structure $\mathfrak{A}$ that satisfies T_1, also satisfies T_2 and vice versa: $\mathfrak{A} \models T_1 \Leftrightarrow \mathfrak{A} \models T_2$.

for equivalence $\alpha \Leftrightarrow (\alpha_1 \vee \cdots \vee \alpha_n)$. Call Convert-To-FO-Clause(α_i, χ_i) for each χ_i and α_i, $i \leq n$.

4. If $\psi = \neg(\chi)$, add Tseitin predicate symbol β to Γ. Generate clauses for $\alpha \Leftrightarrow \neg\beta$ and call Convert-To-FO-Clause(β, χ).

Example. In execution of Convert-To-FO-Clause(ψ, α), for a ground FO formula $\psi = P(1,3) \vee (P(1,1) \wedge P(1,2))$, and Tseitin predicate α, new Tseitin predicates α_1 and α_2 are introduced for sub-formulas $\psi_1 = P(1,3)$ and $\psi_2 = (P(1,1) \wedge P(1,2))$, and then Convert-To-FO-Clause(ψ_1, α_1) and Convert-To-FO-Clause(ψ_2, α_2) are executed, and clauses for $\alpha \Leftrightarrow (\alpha_1 \vee \alpha_2)$ are generated.

- In execution of Convert-To-FO-Clause(ψ_1, α_1), clauses for $\alpha_1 \Leftrightarrow P(1,3)$ are generated.
- In execution of Convert-To-FO-Clause(ψ_2, α_2), new Tseitin predicates α_3 and α_4 are introduced respectively for $P(1,1)$, and $P(1,2)$, and clauses for $\alpha_3 \Leftrightarrow P(1,1)$, $\alpha_4 \Leftrightarrow P(1,2)$, and $\alpha_2 \Leftrightarrow (\alpha_3 \wedge \alpha_4)$ are generated.

By executing procedure Convert-To-FO-Clause for all reduced ground formulas generated, we obtain the grounding in clausal form. Expansion vocabulary ε is extended with Γ. By projection out the interpretation of expansion $0-$ary predicate symbols of Γ, any model for these clauses gives a solution for the original problem.

4.1.2 Transforming a FO Clause to a Propositional Clause

In the standard input syntax of SAT solvers, DIMACS format, propositional variables are Natural numbers. To provide proper inputs for SAT solvers, ground FO atoms in FO clauses generated by Convert-To-FO-Clause, must be mapped to propositional variables by a function from ground-atoms to Natural numbers.

One might create a table for mapping, by inserting a new pair (a, a_p) into the table, for each ground atom a in clauses, and assigning the smallest natural number, not assigned yet, a_p, to it. This approach guarantees we are not introducing propositional variables more than the number of ground atoms that have appeared in ground FO clauses. The drawback is that both look-ups and inserts are done in linear time. To find the propositional variable assigned to each FO ground atom, a look-up is needed. If no entry for the ground atom is found in the mapping table, it means the ground atom has not already been assigned to a propositional variable, and a new entry will be inserted for it. A hashing function may seem to be a good choice for this purpose. But we need to find two hashing functions, both doing look-ups in a constant time: one from ground atoms to propositional variables, and one from propositional variables to ground atoms, that does the inverse of first one. The second one is needed in the ungrounding step when a satisfying assignment is translated to a FO solution. A mapping with these properties should be defined with functions that relate structural properties of a ground atom to a number (propositional variable). As we do not know what ground atoms may appear in the answer of a formula, such relation can not be defined without any gap in the propositional variables. So a hashing functions is not applicable here.

The other approach is to reserve a prefix of Natural numbers for expansion ground atoms of ε, w.r.t. the size of sorts, and use a one-to-one function Map-To-PropVar to assign a unique natural number to each expansion ground atom.
For expansion predicates $P_1, \ldots, P_n$ declared in problem specification file and $D_1, \ldots, D_n$ the maximum size of their interpretation, function Map-To-PropVar can be defined as below:

$$\text{Map-To-PropVar: } \{\text{Expansion Ground Atoms}\} \longrightarrow [1 \cdots \textstyle\sum_{i<n} D_i]$$
$$\text{Map-To-PropVar}(P_m(a_k, \ldots, a_1)) = \textstyle\sum_{i<m} D_i + \sum_{i<k}((Image(a_i)) * \prod_{j<k} DOM(P_{mj})) + 1$$

$DOM(P_{mj})$ denotes the size of domain of jth argument in predicate P_m. $Image(a_i)$ is the natural number associated to the value of a_i in the correspondence of its sort to natural numbers.

We then assign the smallest unreserved and unassigned Natural numbers to Tscitin predicate symbols of Γ. In this approach, look-ups for the ground atoms of ε is done in constant time by the one-to-one mapping defined by function Map-To-PropVar.

We implement the second approach in MXG, and moreover call function Map-To-PropVar in steps of Convert-To-FO-Clause procedure to simultaneously create propositional clauses. Let function Get-Next-PropVar return the next unassigned and unreserved number. Convert-To-PropClause(α, ψ), defined as below, generates propositional clauses equivalent to the ground FO formula ψ. α is a natural number (a propositional variable).

Convert-To-PropClause(α, ψ):

1. If ψ is an atom, generate a unary propositional clause Map-To-PropVar(ψ) and return.

2. If $\psi = \chi_1 \wedge \cdots \wedge \chi_n$, generate propositional clauses for equivalence $\alpha \Leftrightarrow (\alpha_1 \wedge \cdots \wedge \alpha_n)$

where α_i is Map-To-PropVar(χ_i) if χ_i is an atom, or Get-Next-PropVar if χ_i is a non-atom. Call Convert-To-PropClause(α_i, χ_i) for each $\chi_i, i < n$.

3. If $\psi = \chi_1 \vee \cdots \vee \chi_n$, generate propositional clauses for equivalence $\alpha \Leftrightarrow (\alpha_1 \vee \cdots \vee \alpha_n)$ where α_i is Map-To-PropVar(χ_i) if χ_i is an atom, or Get-Next-PropVar if χ_i is a non-atom. Call Convert-To-PropClause(α_i, χ_i) for each $\chi_i, i < n$.

4. If $\psi = \neg(\chi)$, generate clauses for $\alpha \Leftrightarrow \neg\beta$ and call Convert-To-PropClause(β, χ).

Get-Next-PropVar generates the Tseitin variables in this transformation. In procedure Map-To-PropVar the ground FO atoms are looked up in constant time.

4.2 "Un-grounder" module

Propositional assignments to satisfy the CNF clauses produced by Convert-To-PropClause after projecting out the Tseitin variables are, in one-to-one correspondence with solutions of the instance. There is no standard output format for SAT-solvers, which necessiates writing a program for each choice of SAT-solver, to parse the output of the SAT-solver and collect the set of literals giving the truth assignment. If the problem is satisfiable, the set of literals are sent to MXG. Those true literals that are in the range of Map-To-PropVar constitute an interpretation of expansion vocabularies in the founded model. MXG applies the reverse of mapping Map-To-PropVar on true literals in range of Map-To-PropVar to obtain the ground atoms. An interpretation of an expansion predicate, for this solution, is the collection of ground atoms generated for it by this reverse mapping.

If the SAT-solver finds the CNF formula to be unsatisfiable, an empty set is sent to MXG. MXG sends an output message "No Answer", to show that problem has no solution.

4.3 Handling Inductive Definitions

MXG can handle only two fragments of inductive definitions, Horn-IDs and Comp-IDs as defined in Section 2.4.3. In this section we explain how MXG handles the semantics of inductive definitions for these fragments.

4.3.1 Horn-ID

For a Horn-ID, MXG treats $\leftarrow$ as $\supset$. This gives a set of Horn formulas that has a unique minimum model, which is the same as the well-founded model of the inductive definition. The minimum model for a propositional Horn theory can be found by a linear time procedure, by applying unit propagation techniques from SAT [27] [16]. MXG has a built-in module, ComputeMinModel, that precomputes this model. The defined predicate is then treated as an instance predicate for the remainder of the grounding of this specification. For the example of Section 2.4.3, MXG rewrites the rules with the following FO implications and generates reduced ground clauses for them. Then procedure ComputeMinModel is called to find the interpretation of *Dist*.

$\forall v\, n : ((v = Start \wedge n = MIN) \supset Dist(v, n))$

$\forall v\, n : ((\exists u m : (Dist(u, m) \wedge Edge(u, v) \wedge SUCC(m, n))) \supset Dist(v, n))$

4.3.2 Comp-ID

If an inductive definition is not a Horn-ID, MXG replaces it with its *Completion* [10], by interpreting $\leftarrow$ as $\Leftrightarrow$. The substitution is correct if the definition is a Comp-ID, but not in general. For definitions of *Odd* and *Even*, in example of Section 2.4.3, MXG replaces them with their completion.

$\forall n : (Even(n) \Leftrightarrow [n = MIN \vee \neg Odd(n) \vee (\exists n' : (Odd(n') \wedge SUCC(n', n)))])$

$\forall n : (Odd(n) \Leftrightarrow [\neg Even(n) \vee \exists n' : (Even(n') \wedge SUCC(n', n))])$

As the definitions are Comp-ID's, they are equivalent to their completion. In any expansion structure, interpretation of *Odd* is the set of odd numbers in S, and interpretation of *Even* is the set of even numbers in S, which is our intended interpretation of the definitions.

4.4 Handling Cardinality Constraints

There is no standard format for propositional cardinality clauses. MXG uses MXC, a SAT+Cardinality solver, for solving the ground formulas including cardinality constraints. A cardinality clause in MXC is in the following format:

$$\#\ lb\ ub\ l_1\ \ldots\ l_n\ 0$$

It bounds the number of true literals from the set $\{l_1, \ldots, l_n\}$ to be at least lb and at most ub. Here lb and ub are natural numbers including 0.

To ground formula $\forall \overline{x} : \odot(\nu; \overline{y}; \phi(\overline{x}, \overline{y}))$, for each $\overline{a} : \overline{x} \rightarrow A$, a propositional cardinality clause of the following form is generated:

$\#\ lb\ \ ub\ \ [\phi(\overline{a}, \overline{b_1})]\ \ \ldots\ \ [\phi(\overline{a}, \overline{b_k})]\ \ 0$, for all $\overline{b_i} : \overline{y} \rightarrow A$

The notation $\overline{a} : \overline{x} \rightarrow A$, means that each value $a_i, 1 \leq i \leq k$ in tuple $\overline{a} = (a_1, \ldots, a_k)$ is taken from the sort of variable x_i in attributes $\overline{x} = (x_1, \ldots, x_k)$. $[\phi(\overline{a}, \overline{b_i})]$ is the propositional Tseitin variable assigned to $\phi(\overline{a}, \overline{b_i})$ by procedure Get-Next-PropVar. The values of lb, ub are set accordingly for $UB, CARD, LB$:

- $lb = 0, ub = \nu$ for UB,
- $lb = ub = \nu$ for $CARD$,
- $lb = \nu, ub = k$ for LB where k is the size of set $\{\overline{b_i} : \overline{y} \rightarrow A\}$.

Then MXG generates the reduced grounding of FO sentences $\phi(\overline{a}, \overline{b_i})$. and Convert-To-PropClause($[\phi(\overline{a}, \overline{b_i})], \phi(\overline{a}, \overline{b_i})$) is executed.

Cardinality constraints are not only a convenient way for expressing counting properties, but also decrease grounding time noticeably in most of the cases. We observe that MXC grounding time is reduced by using an encoding with cardinality constraints rather than an equivalent one without (See Section 6.4).

Example : The following two FO sentences define predicate Map, over $Num * Num$, to be a one-to-one relation:

$\forall x : \exists y : Map(x, y)$

$\forall x y_1 y_2 : ((Map(x, y_1) \wedge Map(x, y_2)) \supset y_1 = y_2)$

For an instance structure with $Num = [1 \ldots n]$, MXG generates n propositional clauses of size n in the form:

$Map(i, 1) \vee \cdots \vee Map(i, n)$, for $1 \leq i \leq n$,

and $n(n-1)$ binary propositional clauses of the form:

$\neg Map(i, j) \vee \neg Map(i, k)$, for $1 \leq i, j, k \leq n$, $j \neq k$

The one-to-one correspondence property of predicate Map can be defined by a cardinality constraint clause as below:

$\forall x : CARD(1; y; Map(x, y))$

for which MXG generates only n propositional cardinality clauses of size n :

$\#\ 1\ \ 1\ \ Map(i, 1)\ \ \ldots\ \ Map(i, n)\ \ 0$

For large values of n, the differences in size and number of clauses in the first and second axiomatization can be very significant, and it takes more time for MXG to generates clauses of first axiomatization and save them in a file.

4.5 Reducing the Number of Tseitin Variables

Tseitin variables are introduced in transformation of ground FO formulas to propositional clauses, wherever there is an alternation of disjunction and conjunction in the formula. For example 3 Tseitin variables q_1, q_2, q_3 may be needed in a naive transformation of propositional formula $(p_1 \vee p_2) \wedge (p_3 \vee p_4)$ to clausal form: $q_1 \Leftrightarrow q_2 \wedge q_3, q_2 \Leftrightarrow p_1 \vee p_2$, and $q_3 \Leftrightarrow p_3 \wedge p_4$.

Introduction of new Tseitin variables increases the number of clauses, as well as variables, and consequently makes the solving step more complex. It is avoidable in some cases :

- **Direct Handling of a Conjunction Formula:** Propositional formula $S = S_1 \wedge \cdots \wedge S_n$ can be transformed to CNF form by a set of unary clauses $\{S_1, \ldots, S_n\}$ without introducing any Tseitin variables. For the given sample formula, clauses $p_1 \vee p_2$, and $p_3 \vee p_4$, implying the same constraints.

- **Pushing Negation:** Pushing in negations at the beginning of a propositional formula, and transforming the result formula to CNF form, often reduces the number of Tseitin variables needed for transformation. Formula $\neg(p_1 \wedge p_2 \wedge p_3)$ without pushing negation generates $\neg q_1$ and clauses for $q_1 \Leftrightarrow (p_1 \wedge p_2 \wedge p_3)$, while pushing negation inside results to a single clause $\neg p_1 \vee \neg p_2 \vee \neg p_3$.

  For formula $\neg(p_1 \vee p_2 \vee p_3)$, applying the "pushing negation" inside together with "handling the conjunction directly" gives clauses $\neg p_1, \neg p_2$, and $\neg p_3$. These unary clauses are easy for SAT solver to handle, comparing to clauses of larger size generated for $q_1 \Leftrightarrow p_1 \vee p_2 \vee p_3$

- **Merging Consecutive Conjunctions or Disjunctions:** The answer to formula $\exists x P(x) \vee Q(x)$ w.r.t. to the instance structure $\mathfrak{A} = \{A; =\}$ where $A = \{1, 2\}$ is the ground formula:

  $(P(1) \vee Q(1)) \vee (P(1) \vee Q(2)) \vee (P(2) \vee Q(1)) \vee (P(2) \vee Q(2))$.

In the general case MXG replaces each formula in the parenthesis by a new Tseitin variable, and generates clauses $d_1 \vee d_2 \vee d_3 \vee d_4$ and clauses for equivalences of the form $d_1 \Leftrightarrow (P(1) \vee Q(1))$. Detecting that the whole formula is a plain clause and there is no operator alternation allows MXG to generate clauses in a more efficient way. For the given formula it generates just one clause $P(1) \vee Q(1) \vee Q(2) \vee P(2)$, after dropping the repeated variables.

Chapter 5

A Grounding Algorithm for MXG

In this chapter we present in the detail the grounding algorithm of MXG, Gnd-Hidden, for generating a "reduced grounding" of a FO formula w.r.t. a given instance structure. Gnd-Hidden is based on a generalization of relational algebra operations to "extended-hidden" relations. Gnd-Hidden is a variant of Gnd, which uses "extended" relations. Applying relational algebra operations on extended relations was first used in [49] to obtain a theoretically efficient algorithm for k-guarded fragment, GF_k, of FO. Our contribution includes a generalization of the union operator, introduction of division operator, intorduction of extended relations with hidden columns, and producing an efficient implementation.

Note that in a multi-sorted logic, each variable has a specific sort, and the universe A is the collection of all sorts in a problem. Throughout this chapter, the notation $\bar{a} : \bar{x} \longrightarrow A$, means that each value $a_i, 1 \leq i \leq k$ in tuple $\bar{a} = (a_1, \ldots, a_k)$ is taken from the sort of variable x_i in attributes $\bar{x} = (x_1, \ldots, x_k)$ [1].

5.1 Gnd: A Grounding with Extended Relations

Before explaining the grounding algorithm, we need to define some concepts used in the definition of Gnd.

Definitions of this section, and the relational algebra operations except for division, and union were first presented in [49] (In [49], they define a special case of our union).

[1]As we explained in Section 3.2.3, in MXG predicates are typed, and sort of variables are inferred from the role of a variable in formula.

Def.(Extended Relation): An *extended relation* $T_{\overline{x}}$ is a table with attributes $\overline{x}$, and a ground formula attached to each tuple. Each entry in $T_{\overline{x}}$ may be represented by pairs $(\overline{a}, \psi_T(\overline{a}))$, where $\psi_T(\overline{a})$ is the ground formula associated with tuple $\overline{a}$. $\psi_T : \overline{c} \longrightarrow \Phi$ is a function from type elements $\overline{c}$ to FO formulas Φ. Tuples with formula $false$ do not explicitly appear in the representation of an extended table.

Def.(Answer to formula): An extended relation $T_{\overline{x}}$ is an answer to formula $\phi(\overline{x})$ w.r.t. structure $\mathfrak{A}$, iff for all $\overline{a} : \overline{x} \longrightarrow A$ and $\overline{a} \in T_{\overline{x}}$, $\psi_T(\overline{a})$ is a reduced grounding of $\phi(\overline{a})$. The answer for an atomic formula $P(\overline{x})$ is obtained and precomputed from:

- If P is an instance predicate, an extended relation $T_{\overline{x}}$ with tuples from interpretation of P and formula *true* attacheded to each tuple.
- If P is an expansion predicate, an extended relation $T_{\overline{x}}$ with tuples $(\overline{a}, P(\overline{a}))$ for all $\overline{a} : \overline{x} \longrightarrow A$.

The answer for a sentence is an extended relation containing only the empty tuple; the formula associated with that tuple is the reduced grounding of the sentence.

Example 1 *The answer to formulas $P(x,y)$, and $Q(x,z)$ w.r.t. instance structure $\mathfrak{A} = (A; P^{\mathfrak{A}})$, where A has only one sort $D = [1 \ldots 3]$, and $P^{\mathfrak{A}} = \{(1,2),(2,3)\}$, is represented in Figures 5.1 and 5.2 by extended relations $R_{x,y}, S_{x,z}$.*

x	y	$\psi_R(x,y)$
1	2	$True$
2	3	$True$

Figure 5.1: $R_{x,y}$: Answer to $P(x,y)$ w.r.t. structure $\mathfrak{A}$

5.1.1 An algebra for extended relations

The standard relational algebra operations can be generalized to extended relations. Here we give the definition of join, union, complement, projection, and division on extended relations. If no entry for tuple $\overline{a}$ exists in an extended table, it is taken as $(\overline{a}, false)$ for operations involving that table.

x	z	$\psi_S(x,z)$
1	1	$Q(1,1)$
1	2	$Q(1,2)$
1	3	$Q(1,3)$
2	1	$Q(2,1)$
2	2	$Q(2,2)$
2	3	$Q(2,3)$
3	1	$Q(3,1)$
3	2	$Q(3,2)$
3	3	$Q(3,3)$

Figure 5.2: $S_{x,z}$: Answer to $Q(x,z)$ w.r.t. structure $\mathfrak{A}$

Def.(Extended Join): $T_{\overline{x}} = R_{\overline{y}} \bowtie S_{\overline{z}}$, where $\overline{x} = \overline{y} \cup \overline{z}$, iff for any $(\overline{a}, \psi_T(\overline{a})) \in T_{\overline{x}}$: $(\overline{a}|\overline{y}, \psi_R(\overline{a}|\overline{y})) \in R_{\overline{y}}$, $(\overline{a}|\overline{z}, \psi_S(\overline{a}|\overline{z})) \in S_{\overline{z}}$, $\psi_T(\overline{a}) = \psi_R(\overline{a}|\overline{y}) \wedge \psi_S(\overline{a}|\overline{z})$.

Proposition: $T_{\overline{x}} = R_{\overline{y}} \bowtie S_{\overline{z}}$ is the answer to $\phi_R(\overline{y}) \wedge \phi_S(\overline{z})$ w.r.t. $\mathfrak{A}$, where $R_{\overline{y}}$ and $S_{\overline{z}}$ are the answers to $\phi_R(\overline{y})$ and $\phi_S(\overline{z})$ with respect to structure $\mathfrak{A}$.

Proof. $R_{\overline{y}}$ is the answer to $\phi_R(\overline{y})$ w.r.t. structure $\mathfrak{A} = (A; \sigma^{\mathfrak{A}})$, which means for any expansion of $\mathfrak{A}$ to structure $\mathfrak{B} = (A, \sigma^{\mathfrak{A}}, \varepsilon^{\mathfrak{B}})$, $\mathfrak{B} \models \phi_R(\overline{y})$ iff $\mathfrak{B} \models \psi_R(\overline{b})$ for all $(\overline{b}, \psi_R(\overline{b}))$. The same argument for $S_{\overline{z}}$ states that for any structure $\mathfrak{B}$, an expansion of $\mathfrak{A}$, $\mathfrak{B} \models \phi_S(\overline{z})$ iff $\mathfrak{B} \models \psi_S(\overline{c})$ for all $(\overline{c}, \psi_S(\overline{c}))$. $\psi_R(\overline{b}$ and $\psi_S(\overline{c}$ are the formulas attached to tuples with values $\overline{b}$, $\overline{c}$ respectively in extended tables $R_{\overline{y}}$ and $S_{\overline{z}}$.

On the other hand, for any structure $\mathfrak{B}$, $\mathfrak{B} \models \psi_1 \wedge \psi_2$ iff $\mathfrak{B} \models \psi_1$, and $\mathfrak{B} \models \psi_2$ (by the semantics of predicate calculus). Each tuple $(\overline{a}, \psi_T(\overline{a})) \in T_{\overline{x}}$ is constructed from two tuples $(\overline{b}, \psi_R(\overline{b})) \in R_{\overline{y}}$, $(\overline{c}, \psi_S(\overline{c})) \in S_{\overline{z}}$. Thus for any structure $\mathfrak{B}$, $\mathfrak{B} \models \psi_R(\overline{b})$ and $\mathfrak{B} \models \psi_S(\overline{c})$ iff $\mathfrak{B} \models \psi_T(\overline{a})$, for all $(\overline{a}, \psi_T(\overline{a})) \in T_{\overline{x}}$, $(\overline{b}, \psi_R(\overline{b})) \in R_{\overline{y}}$, $(\overline{c}, \psi_S(\overline{c})) \in S_{\overline{z}}$. The equality defines $T_{\overline{x}}$ as the answer to $\phi_T(\overline{x}) = \phi_R(\overline{y}) \wedge \phi_S(\overline{z})$ given $R_{\overline{y}}$ and $S_{\overline{z}}$ are answers to $\phi_R(\overline{y})$ and $\phi_S(\overline{z})$ w.r.t. $\mathfrak{A}$.■

Example 2 *For $P, Q, \mathfrak{A}$ as defined in Example 1, the answer to formula $P(x,y) \wedge Q(x,z)$ w.r.t. $\mathfrak{A}$ is computed by joining extended tables $R_{x,y}$ and $S_{y,z}$ of Figures 5.1 and 5.2. $T_{x,y,z} = R_{x,y} \bowtie S_{x,z}$ is shown in Figure 5.3.*

Def.(Extended Union): $T_{\overline{x}} = R_{\overline{y}} \cup S_{\overline{z}}$, where $\overline{x} = \overline{y} \cup \overline{z}$, iff for any $(\overline{a}, \psi_T(\overline{a})) \in T_{\overline{x}}$, $(\overline{a}|\overline{y}, \psi_R(\overline{a}|\overline{y})) \in R_{\overline{y}}$, $(\overline{a}|\overline{z}, \psi_S(\overline{a}|\overline{z})) \in S_{\overline{z}}$, $\psi_T(\overline{a}) = \psi_R(\overline{a}|\overline{y}) \vee \psi_S(\overline{a}|\overline{z})$.

x	y	z	$\psi_T(x,y,z)$
1	2	1	$Q(1,1)$
1	2	2	$Q(1,2)$
1	2	3	$Q(1,3)$
2	3	1	$Q(2,1)$
2	3	2	$Q(2,2)$
2	3	3	$Q(2,3)$

Figure 5.3: $T_{x,y,z}$: Answer to formula $P(x,y) \wedge Q(x,z)$ for Example 1

Proposition: $T_{\overline{x}} = R_{\overline{y}} \cup S_{\overline{z}}$ is the answer to $\phi_R(\overline{y}) \vee \phi_S(\overline{z})$ w.r.t. $\mathfrak{A}$, where $R_{\overline{y}}$ and $S_{\overline{z}}$ are answers to $\phi_R(\overline{y}), \phi_S(\overline{z})$ with respect to structure $\mathfrak{A}$.

Example 3 *For $P, Q, \mathfrak{A}$ as defined in Example 1, the answer to formula $P(x,y) \vee Q(x,z)$ w.r.t. $\mathfrak{A}$, is the union of extended tables $R_{x,y}$, and $S_{y,z}$, of Figures 5.1 and 5.2. $W_{x,y,z} = R_{x,y} \cup S_{x,z}$ is shown in Figure 5.4.*

x	y	z	$\psi_W(x,y,z)$
1	1	1	$Q(1,1)$
1	1	2	$Q(1,2)$
1	1	3	$Q(1,3)$
1	2	1	*true*
1	2	2	*true*
1	2	3	*true*
1	3	1	$Q(1,1)$
1	3	2	$Q(1,2)$
..	..	..	.....
2	3	1	*true*
2	3	2	*true*
2	3	3	*true*
3	1	1	$Q(3,1)$
..	..	..	.....
3	3	3	$Q(3,3)$

Figure 5.4: $W_{x,y,z}$: Answer to formula $P(x,y) \vee Q(x,z)$ for Example 1

Def.(Extended Complement): $T_{\overline{x}} = \overline{R_{\overline{x}}}$, iff for any $(\overline{a}, \psi_T(\overline{a})) \in T_{\overline{x}}$, $(\overline{a}, \neg\psi_T(\overline{a})) \in R_{\overline{x}}$.

Proposition: $T_{\overline{x}} = \overline{R_{\overline{x}}}$ is the answer to $\neg\phi_R(\overline{x})$ w.r.t. $\mathfrak{A}$, where $R_{\overline{x}}$ is the answer to $\phi_R(\overline{x})$ w.r.t. structure $\mathfrak{A}$.

Def.(Extended Projection): $T_{\overline{x'}} = \Pi_{\overline{x'}} R_{\overline{x}}$, for $\overline{x'} \subset \overline{x}$, iff for any tuple $(\overline{a'}, \psi_T(\overline{a'})) \in T_{\overline{x'}}$, there are tuples $(\overline{a}, \psi_R(\overline{a})) \in R_{\overline{x}}$ such that $\overline{a}|\overline{x'} = \overline{a'}$, and $\psi_T(\overline{a'}) = \bigvee_{(\overline{a}, \psi_R(\overline{a})) \in R_{\overline{x}}} \psi_R(\overline{a})$.

Proposition:$T_{\overline{x'}} = \Pi_{\overline{x}\setminus\overline{x'}} R_{\overline{x}}$ is the answer to $\exists \overline{x'} \phi_R(\overline{x})$ w.r.t. $\mathfrak{A}$, where $R_{\overline{x}}$ is the answer to $\phi_R(\overline{x})$ w.r.t. structure $\mathfrak{A}$.

Example 4 *For $P, Q, \mathfrak{A}$ as defined in Example 1, the answer to formula $\exists\ z\ :\ P(x,y) \wedge Q(x,z)$ w.r.t. $\mathfrak{A}$, is obtained by the projection of $T_{x,y,z}$ of Figure 5.3, answer to $P(x,y) \wedge Q(x,z)$ w.r.t. $\mathfrak{A}$, on attributes x, y. $V_{x,y} = \Pi_z T_{x,y,z}$ is shown in Figure 5.5.*

x	y	$\psi_V(x,y)$
1	2	$Q(1,1) \vee Q(1,2) \vee Q(1,3)$
2	3	$Q(2,1) \vee Q(2,2) \vee Q(2,3)$

Figure 5.5: $V_{x,y}$: Answer to formula $\exists\ z\ :\ P(x,y) \wedge Q(x,z)$ w.r.t. structure $\mathfrak{A}$

Def.(Extended Division): $T_{\overline{x''}} = R_{\overline{x}}/\{\overline{x'} \rightarrow A\}$, where $x'' = \overline{x} \setminus \overline{x'}$ iff for any tuple $(\overline{a''}, \psi_T(\overline{a''})) \in T_{\overline{x''}}$, and for all $\overline{a'} : \overline{x'} \rightarrow A$, there are tuples $(\overline{a''} \cup \overline{a'}, \psi_R(\overline{a''} \cup \overline{a'})) \in R_{\overline{x}}$ and $\psi_T(\overline{a''}) =$
$\bigwedge_{(\overline{a''} \cup \overline{a'}, \psi_T(\overline{a''} \cup \overline{a'})) \in R_{\overline{x}}} \psi_R(\overline{a''} \cup \overline{a'})$.

Proposition: $T_{\overline{x}\setminus\overline{x'}} = R_{\overline{x}}/\{\overline{x'} \rightarrow A\}$ is the answer to $\forall \overline{x'} \phi_R(\overline{x})$ w.r.t. $\mathfrak{A}$, where $R_{\overline{x}}$ is the answer to $\phi_R(\overline{x})$ w.r.t. structure $\mathfrak{A}$.

Example 5 *For $V_{x,y}$ computed in Example 4, the answer to formula $\forall\ x\ y\ :\ V_{x,y}$ w.r.t. $\mathfrak{A}$ is obtained by dividing $V_{x,y}$ on $\{(x,y)\} = \{(1,1), \ldots, (3,3)\}$. $U_\emptyset = V_{x,y}/\{x,y\}$ is empty, and can be represented by a table with one entry $(\emptyset, false)$.*

5.1.2 Algorithm Gnd

Now we can give the definition of procedure *Gnd*$(\phi, \mathfrak{A})$. The input is a FO formula ϕ and an instance structure $\mathfrak{A} = (A, \sigma^{\mathfrak{A}})$. It inductively operates on the structure of ϕ and computes answers to sub-formulas of ϕ w.r.t. $\mathfrak{A}$. Its output is an extended table which is the answer to formula ϕ w.r.t. structure $\mathfrak{A}$. The cases are:

1. $\phi = P(\overline{x})$, P instance: returns $T_{\overline{x}}$ with tuples $\{(\overline{a}, true) : \overline{a} \in P^{\mathfrak{A}}\}$,
2. $\phi = P(\overline{x})$, P expansion: returns $T_{\overline{x}}$ with tuples $\{(\overline{a}, P(\overline{a})) : \overline{a} : \overline{x} \to A\}$,
3. $\phi = \delta \wedge \psi$: $Gnd(\delta, \mathfrak{A}) \bowtie Gnd(\psi, \mathfrak{A})$,
4. $\phi = \delta \vee \psi$: $Gnd(\delta, \mathfrak{A}) \cup Gnd(\psi, \mathfrak{A})$,
5. $\phi = \neg\psi$: $\overline{Gnd(\psi, \mathfrak{A})}$,
6. $\phi = \exists\overline{y}\psi(\overline{x})$: $\Pi_{\overline{x}\setminus\overline{y}} Gnd(\psi(\overline{x}), \mathfrak{A})$,
7. $\phi = \forall\overline{y}\psi(\overline{x})$: $Gnd(\psi(\overline{x}), \mathfrak{A}) / \{\overline{y} \to A\}$.

Theorem 1 *For any given FO formula ϕ and structure $\mathfrak{A}$, Gnd returns an answer to ϕ w.r.t. $\mathfrak{A}$.*

Proof: Proof is by induction on the structure of ϕ. The base case is when ϕ is an atom $P(\overline{x})$:

- For P being an instance symbol, Gnd returns an extended relation $R_{\overline{x}}$ with tuples $\{(\overline{a}, true) : \overline{a} \in P^{\mathfrak{A}}\}$. $R_{\overline{x}}$ is the answer to $P(\overline{x})$, as the formula attached to each tuple in $R_{\overline{x}}$ is the reduced grounding of $P(\overline{x})$ w.r.t. $\mathfrak{A}$.
- For P being an expansion symbol, Gnd returns an extended relation $S_{\overline{x}}$ with tuples $\{(\overline{a}, P(\overline{a})) : \overline{a} : \overline{x} \to A\}$. $S_{\overline{x}}$ is the answer to $P(\overline{x})$, as the formula attached to each tuple in $S_{\overline{x}}$ is the reduced grounding of $P(\overline{x})$ w.r.t. $\mathfrak{A}$.

The induction step follows from the propositions in Section 5.1.1. Gnd operates inductively on the structure of ϕ and at each step computes a certain relational algebra operation corresponded to each logical operation. According to propositions of the Section 5.1.1, the result of each operation is the answer to that subformula w.r.t. $\mathfrak{A}$. Thus finally Gnd returns answer to ϕ w.r.t. $\mathfrak{A}$.■

5.2 Gnd-Hidden: Grounding With Hidden Variables

Gnd in case 2 constructs a 'universal' extended relation over variables $\bar{x}$ for each occurance of an expansion predicate P. The size of such a relation for an expansion predicate that is defined over large domains may be huge, and it may contribute greatly to the execution time of Gnd. Gnd-Hidden is a refinement of Gnd based on *extended-hidden* relations, for which columns corresponding to a universal inclusion are left implicit. Definition of Gnd-Hidden is the same as Gnd except that it applies the relational algebra operations adapted to extended-hidden relations. In following we give definitions of reduced grounding, answer to a formula, and relational algebra operations for extended-hidden relations.

In the following definitions, for a $\sigma \cup \varepsilon$-formula ϕ (σ is the instance vocabulary and ε is the expansion vocabulary), we denote the set of free variables in ϕ that are arguments only to predicate symbols of ε by $ExpVar(\phi)$, and the remaining free variables by $InstVar(\phi)$. Without loss of generality, and as a convention, we denote formula ϕ with free variables $InstVar(\phi) \cup ExpVars(\phi)$ by $\phi(InstVar(\phi); ExpVar(\phi))$.

Def.(Reduced-Hidden Grounding): Let $\phi(\bar{x};\bar{y})$ be a formula over vocabulary $\sigma \cup \varepsilon$. A *reduced-hidden grounding* of $\phi(\bar{x};\bar{y})$ w.r.t. $\mathfrak{A}$ is a formula $\psi(\bar{y})$ over vocabulary $\varepsilon \cup \tilde{A}$ only, such that for any structure $\mathfrak{B} = (A; \sigma^{\mathfrak{A}}; \tilde{A}^{\mathfrak{A}^*}, \epsilon^{\mathfrak{B}})$, and any object assignment ω, $\mathfrak{B}[\omega] \models \phi(\bar{x};\bar{y})$ iff $\mathfrak{B}[\omega] \models \psi(\bar{y})$.

For a FO sentence with no free variables, a reduced-hidden grounding is also a reduced grounding. MXG obtains a reduced-hidden grounding of a formula $\phi(\bar{x};\bar{y})$ by instantiating only variables $\bar{x}$ and simultanously 'evaluating out' the instance vocabulary.

Def.(Extended-Hidden relation): An extended-hidden relation $T^{\bar{y}}_{\bar{x}}$ is an extended relation with explicit attributes $\bar{x}$ and hidden attributes $\bar{y}$. Values of hidden attributes do not appear explicitly in tuples. $\psi_T(\bar{a})$ for each pair $(\bar{a}, \psi_T(\bar{a}))$, $\bar{a} : \bar{x} \rightarrow A$, is a ground formula with free variables $\bar{y}$. Extended-hidden table $T^{\bar{y}}_{\bar{x}}$, with entries $(\bar{a}, \psi_T(\bar{a}))$, $\bar{a} : \bar{x} \rightarrow A$, is a compact representation of extended table $T_{\bar{x}\bar{y}}$ with entries $((\bar{a}, \bar{b}), \psi_T(\bar{a})(\bar{b}/\bar{y}))$ where $\bar{a} : \bar{x} \rightarrow A$, $\bar{b} : \bar{y} \rightarrow A$.

Extended relation $T_{\bar{x}}$ is an extended-hidden relation with empty set of hidden attributes, $T^{\emptyset}_{\bar{x}}$.

Def.(Universal Extended-Hidden Relation): A universal extended-hidden relation for explicit attributes $\bar{x}$, and hidden attributes $\bar{y}$, w.r.t. $\mathfrak{A}$, is a table with pairs $(\bar{a}, \mathit{true}), \bar{a} : \bar{x} \rightarrow A$, and is represented by $\widehat{U}^{\bar{y}}_{\bar{x}}$.

Def.(Answer to formula): The extended-hidden relation $T_{\overline{x}}^{\overline{y}}$ is the answer to formula $\phi(\overline{x};\overline{y})$ w.r.t. structure $\mathfrak{A}$, iff for all $\overline{a} : \overline{x} \to A$, $\psi_T(\overline{a})$ is a reduced-hidden grounding for $\phi(\overline{x};\overline{y})$.

The answer for an atomic formula $P(\overline{x})$ is obtained and precomputed from:

- If P is an instance predicate, an extended-hidden relation $T_{\overline{x}}^{\emptyset}$ with tuples from interpretation of P and formula *true* attacheded to each tuple.
- If P is an expansion predicate, an extended-hidden relation $T_{\emptyset}^{\overline{x}}$ with one tuple $(\emptyset, P(\overline{x}))$.

Example 6 *For $P, Q, \mathfrak{A}$ as defined in Example 1, the answer to $P(x,y)$, $R_{x,y}^{\emptyset}$, is the same as extended table shown in Figure 5.1. The answer to $Q(x,z)$, $S_{\emptyset}^{x,z}$, is a table with one tuple as shown below:*

$\emptyset$	ψ_S
	$Q(x,z)$

5.2.1 An Algebra For Extended Hidden Relations

Def.(Extended Join): $T_{\overline{x_t}}^{\overline{y_t}} = R_{\overline{x_r}}^{\overline{y_r}} \bowtie S_{\overline{x_s}}^{\overline{y_s}}$, where $\overline{x_t} = \overline{x_r} \cup \overline{x_s}$ and $\overline{y_t} = (\overline{y_r} \cup \overline{y_s}) \setminus \overline{x_t}$, iff for any tuple $(\overline{a}, \psi_T(\overline{a})) \in T_{\overline{x_t}}^{\overline{y_t}}$, $(\overline{a}|\overline{x_r}, \psi_R(\overline{a}|\overline{x_r})) \in R_{\overline{x_r}}^{\overline{y_r}}$, $(\overline{a}|\overline{x_s}, \psi_S(\overline{a}|\overline{x_s})) \in S_{\overline{x_s}}^{\overline{y_s}}$, and $\psi_T(\overline{a}) = \psi_R(\overline{a}|\overline{x_r}) \wedge \psi_S(\overline{a}|\overline{x_s})$.

Proposition: $T_{\overline{x_t}}^{\overline{y_t}} = R_{\overline{x_r}}^{\overline{y_r}} \bowtie S_{\overline{x_s}}^{\overline{y_s}}$ is the answer to $\phi_R(\overline{x_r};\overline{y_r}) \wedge \phi_S(\overline{x_s};\overline{y_s})$ w.r.t. $\mathfrak{A}$, where $R_{\overline{x_r}}^{\overline{y_r}}$ and $S_{\overline{x_s}}^{\overline{y_s}}$ are answer's to $\phi_R(\overline{x_r};\overline{y_r})$, and $\phi_S(\overline{x_s};\overline{y_s})$ w.r.t. structure $\mathfrak{A}$.

Proof. $R_{\overline{x_r}}^{\overline{y_r}}$ is answer to $\phi_R(\overline{x_r};\overline{y_r})$ w.r.t. structure $\mathfrak{A} = (A; \sigma^{\mathfrak{A}})$, which means that for any expansion of $\mathfrak{A}$ to structure $\mathfrak{B} = (A, \sigma^{\mathfrak{A}}, \varepsilon^{\mathfrak{B}})$, and any object assignment ω, $\mathfrak{B}[\omega] \models \phi_R(\overline{x_r};\overline{y_r})$ iff $\mathfrak{B}[\omega] \models \psi_R(\overline{b})$ for all $(\overline{b}, \psi_R(\overline{b})) \in R_{\overline{x_r}}^{\overline{y_r}}$. The same argument for $S_{\overline{x_s}}^{\overline{y_s}}$ states that for any structure $\mathfrak{B}$, and any object assignment ω, $\mathfrak{B}[\omega] \models \phi_S(\overline{x_s};\overline{y_s})$ iff $\mathfrak{B}[\omega] \models \psi_S(\overline{c})$ for all $(\overline{c}, \psi_S(\overline{c})) \in S_{\overline{x_s}}^{\overline{y_s}}$.

According to the semantics of predicate calculus, for any structure $\mathfrak{B}$, and any object assignment ω, $\mathfrak{B}[\omega] \models \psi_1$ and $\mathfrak{B}[\omega] \models \psi_2$ iff $\mathfrak{B}[\omega] \models \psi_1 \wedge \psi_2$.

Each tuple $(\overline{a}, \psi_T(\overline{a})) \in T_{\overline{x_t}}^{\overline{y_t}}$ is constructed from two tuples $(\overline{b}, \psi_R(\overline{b})) \in R_{\overline{x_r}}^{\overline{y_r}}$ and $(\overline{c}, \psi_S(\overline{c})) \in S_{\overline{x_s}}^{\overline{y_s}}$. Thus for any structure $\mathfrak{B}$, and any object assignment ω, $\mathfrak{B}[\omega] \models \psi_R(\overline{b})$ and $\mathfrak{B}[\omega] \models \psi_S(\overline{c})$ iff $\mathfrak{B}[\omega] \models \psi_T(\overline{a})$, for all $(\overline{a}, \psi_T(\overline{a})) \in T_{\overline{x_t}}^{\overline{y_t}}$, $(\overline{b}, \psi_R(\overline{b})) \in R_{\overline{x_r}}^{\overline{y_r}}$, $(\overline{c}, \psi_S(\overline{c})) \in S_{\overline{x_s}}^{\overline{y_s}}$. The equality defines $T^{\overline{y_t}}_{\overline{x_t}}$ as the answer to $\phi_T(\overline{x_t};\overline{y_t}) = \phi_R(\overline{x_r};\overline{y_r}) \wedge \phi_S(\overline{x_s};\overline{y_s})$ given $R^{\overline{y_r}}_{\overline{x_r}}$ and $S^{\overline{y_s}}_{\overline{x_s}}$ answers to $\phi_R(\overline{x_r};\overline{y_r})$ and $\phi_S(\overline{x_s};\overline{y_s})$ w.r.t. $\mathfrak{A}$.■

Example 7 *For $P, Q, \mathfrak{A}$ as defined in Example 1, the answer to formula $P(x,y) \wedge Q(x,z)$ is the union of $R^{\emptyset}_{x,y}$, and $S^{x,z}_{\emptyset}$ of Example 6. $T^{z}_{x,y} = R^{\emptyset}_{x,y} \bowtie S^{x,z}_{\emptyset}$ is shown in Figure 5.6.*

x	y	$\psi_T(x,y)$
1	2	$Q(1,z)$
2	3	$Q(2,z)$

Figure 5.6: $T^{z}_{x,y}$: Answer to formula $P(x,y) \wedge Q(x,z)$ for Example 1

Def.(Extended Union): $T^{\overline{y_t}}_{\overline{x_t}} = R^{\overline{y_r}}_{\overline{x_r}} \cup S^{\overline{y_s}}_{\overline{x_s}}$, where $\overline{x_t} = \overline{x_r} \cup \overline{x_s}$, $\overline{y_t} = (\overline{y_r} \cup \overline{y_s}) \setminus \overline{x_t}$, iff for any $(\overline{a}, \psi_T(\overline{a})) \in T^{\overline{y_t}}_{\overline{x_t}}$, $(\overline{a}|\overline{x_r}, \psi_R(\overline{a}|\overline{x_r})) \in R^{\overline{y_r}}_{\overline{x_r}}$, $(\overline{a}|\overline{x_s}, \psi_S(\overline{a}|\overline{x_s})) \in S^{\overline{y_s}}_{\overline{x_s}}$, and $\psi_T(\overline{a}) = \psi_R(\overline{a}|\overline{x_r}) \vee \psi_S(\overline{a}|\overline{x_s})$.
Proposition: $T^{\overline{y_t}}_{\overline{x_t}} = R^{\overline{y_r}}_{\overline{x_r}} \cup S^{\overline{y_s}}_{\overline{x_s}}$ is the answer to $\phi_R(\overline{x_r};\overline{y_r}) \vee \phi_S(\overline{x_s};\overline{y_s})$ w.r.t. $\mathfrak{A}$, where $R^{\overline{y_r}}_{\overline{x_r}}$, and $S^{\overline{y_s}}_{\overline{x_s}}$ are answers to $\phi_R(\overline{x_r};\overline{y_r})$ and $\phi_S(\overline{x_S};\overline{y_s})$ w.r.t. structure $\mathfrak{A}$.

Example 8 *For $P, Q, \mathfrak{A}$ as defined in Example 1, the answer to formula $P(x,y) \vee Q(x,z)$ is computed by joining $R^{\emptyset}_{x,y}$, and $S^{x,z}_{\emptyset}$ from Example 6. $W^{z}_{x,y} = R^{\emptyset}_{x,y} \cup S^{x,z}_{\emptyset}$ is shown in Figure 5.7. Note that, in Figure 5.4, the answer to the same formula has 27 tuples.*

Def.(Extended Complement): $T^{\overline{y}}_{\overline{x}} = \overline{R^{\overline{y}}_{\overline{x}}}$ iff for any $(\overline{a}, \psi_T(\overline{a})) \in T^{\overline{y}}_{\overline{x}}$, $(\overline{a}, \neg\psi_T(\overline{a})) \in R^{\overline{y}}_{\overline{x}}$.
Proposition: $T^{\overline{y}}_{\overline{x}} = \overline{R^{\overline{y}}_{\overline{x}}}$ is the answer to $\neg\phi_R(\overline{x};\overline{y})$ w.r.t. $\mathfrak{A}$, where $R^{\overline{y}}_{\overline{x}}$ is the answer to $\phi_R(\overline{x};\overline{y})$ w.r.t. structure $\mathfrak{A}$.
Def.(Extended Projection): $T^{\overline{y'}}_{\overline{x'}} = \Pi_{\overline{x'}\overline{y'}} R^{\overline{y}}_{\overline{x}}$, for $\overline{x'} \subseteq \overline{x}$, and $\overline{y'} \subseteq \overline{y}$, iff for any pair $(\overline{a'}, \psi_T(\overline{a'})) \in T^{\overline{y'}}_{\overline{x'}}$, there are tuples $(\overline{a''}, \psi_S(\overline{a''})) \in S^{\overline{y'}}_{\overline{x''}}$, where $S^{\overline{y'}}_{\overline{x''}} = R^{\overline{y}}_{\overline{x}} \bowtie \widehat{U}^{\emptyset}_{\overline{y''}}$ and $\overline{x''} = \overline{x} \cup \overline{y''}$, $\overline{y''} = \overline{y} \setminus \overline{y'}$, such that $\overline{a''}|\overline{x'} = \overline{a'}$, $\psi_T(\overline{a'}) = \bigvee_{(\overline{a''},\psi_S(\overline{a''})) \in S^{\overline{y'}}_{\overline{x''}}} \psi_S(\overline{a''})$.
Proposition: $T^{\overline{y'}}_{\overline{x'}} = \Pi_{\overline{x'}\overline{y'}} R^{\overline{y}}_{\overline{x}}$, for $\overline{x''} = \overline{x} \setminus \overline{x'}$, and $\overline{y''} = \overline{y} \setminus \overline{y'}$ is the answer to $\exists x''y''\phi_R(\overline{x};\overline{y})$ w.r.t. $\mathfrak{A}$, where $R^{\overline{y}}_{\overline{x}}$ is the answer to $\phi_R(\overline{x};\overline{y})$ w.r.t. structure $\mathfrak{A}$.
Def.(Extended Division): $T^{\overline{y''}}_{\overline{x''}} = R^{\overline{y}}_{\overline{x}} / \{(\overline{x'}, \overline{y'}) \to A\}$, for $\overline{x''} = \overline{x} \setminus \overline{x'}$, and $\overline{y''} = \overline{y} \setminus \overline{y'}$, iff for any pair $(\overline{a''}, \psi_T(\overline{a''})) \in T^{\overline{y''}}_{\overline{x''}}$, and $\overline{a'} : \overline{x'} \to A$, there is a tuple $(\overline{a'''}, \psi_S(\overline{a'''})) \in S^{\overline{y''}}_{\overline{x'''}}$

x	y	$\psi_W(x,y)$
1	1	$Q(1,z)$
1	2	*true*
1	3	$Q(1,z)$
2	1	$Q(2,z)$
2	2	$Q(2,z)$
2	3	*true*
3	1	$Q(3,z)$
3	2	$Q(3,z)$
3	3	$Q(3,z)$

Figure 5.7: $W^z_{x,y}$: Answer to formula $P(x,y) \vee Q(x,z)$

where $S^{\overline{y''}}_{\overline{x'''}} = R^{\overline{y}}_{\overline{x}} \bowtie \widehat{U}^{\emptyset}_{\overline{y'}}$, $\overline{x'''} = \overline{x} \cup \overline{y'}$, such that $\overline{a'''}|\overline{x''} = \overline{a''}$, $\overline{a'''}|\overline{x'} = \overline{a'}$, $\psi_T(\overline{a''}) = \bigwedge_{(\overline{a'''},\psi_S(\overline{a'''}))\in S^{\overline{y''}}_{\overline{x'''}}} \psi_S(\overline{a'''})$.

Proposition: $T^{\overline{y''}}_{\overline{x''}} = R^{\overline{y}}_{\overline{x}}/\{(\overline{x'},\overline{y'}) \to A\}$, for $\overline{x''} = \overline{x} \setminus \overline{x'}$, and $\overline{y''} = \overline{y} \setminus \overline{y'}$ is the answer to $\forall \overline{x'}\overline{y'}\phi_R(\overline{x};\overline{y})$ w.r.t. $\mathfrak{A}$, where $R^{\overline{y}}_{\overline{x}}$ is the answer to $\phi_R(\overline{x};\overline{y})$ w.r.t. structure $\mathfrak{A}$.

5.2.2 Gnd-Hidden Algorithm

The specification of Gnd-Hidden is exactly the same as Gnd, except that relational algebra operations are operating on extended-hidden relations.

For each axiom ϕ in problem theory T, MXG first parses ϕ and builds a syntax tree for it. Each vertex v in the syntax tree has an extended-hidden relation $Ans(v)$. $Ans(v)$ keeps the answer to the sub-formula rooted at v w.r.t. $\mathfrak{A}$. The answers to atomic formulas, located in leaves of the syntax tree, are precomputed by MXG,(See definition of "answer to formula"). The extended-hidden relation for internal vertices will be computed during the execution of algorithm. Gnd is a bottom-up algorithm: starting with answer of atoms in leaves of syntax tree, and applying an appropriate extended-hidden relational algebra operation at each vertex coming up the syntax tree. The answer for the whole formula is in $Ans(v)$, for v is the root of syntax tree.

Theorem 2 *For any given FO formula ϕ and structure $\mathfrak{A}$, Gnd-Hidden returns an answer to ϕ w.r.t. $\mathfrak{A}$.*

Proof: Proof is the same as proof of Theorem 1 by induction on the structure of ϕ and propositions in Section ??. ■

5.3 Complexity of Gnd-Hidden Algorithm

Theorem 3 *For a given FO formula ϕ and a structure $\mathfrak{A}$, Gnd-Hidden runs in time $O(|\mathfrak{A}|^{|\phi|})$.*

Proof. The time and space complexity of each of the relational operations in Gnd-Hidden procedure on two extended hidden relations $R^{\overline{y_R}}_{\overline{x_R}}$ and $T^{\overline{y_T}}_{\overline{x_T}}$ are (the size of an extended-hidden relation is at most $|\mathfrak{A}|$):

- $R^{\overline{y_R}}_{\overline{x_R}} \bowtie T^{\overline{y_T}}_{\overline{x_T}}$ creates an extended-hidden relation of size $O(|\mathfrak{A}|^2)$ in time of order $O(|\mathfrak{A}|^2)$. The same result holds for operation Union.
- $\overline{R^{\overline{y_R}}_{\overline{x_R}}}$ creates an extended-hidden relation of size $O(|\mathfrak{A}|)$ in time of order $O(|\mathfrak{A}|)$.
- $\Pi_{\overline{z}} R^{\overline{y_R}}_{\overline{x_R}}$, returns an extended-hidden relation with $O(|\mathfrak{A}|)$ tuples, and the formula attached to each tuple be a disjnuction of size $|\mathfrak{A}|$. So the size of result is of $O(|\mathfrak{A}|^2)$ and is computed in time of $O(|\mathfrak{A}|^2)$. The same result holds for operation division.

Procedure Gnd-Hidden is called for each logical operator in the formula ϕ. It is executed $O(|\phi|)$ times. Thus the time and space complexity of Gnd-Hidden for computing answer of ϕ w.r.t. structure $\mathfrak{A}$ is $O(|\mathfrak{A}|^{|\phi|})$. ■

For any fixed formula, Gnd-Hidden is a polynomial time reduction from FO-MX to SAT. The number of *Tseitin* variables that may be introduced, in the worst case, are of order $|\mathfrak{A}|^{|\phi|}$. Operations involving only expansion predicate symbols are computed in a constant time. The number of instance predicates symbols in a formula contributes significantly in the computation time. In the worst case, tables of size $|\mathfrak{A}|$ are generated for each predicate, and the actual grounding time is of order $|\mathfrak{A}|^{|\phi|}$. But in practice, usually this does not happen, and the actual grounding time is less, as the number of tuples involving in computation are less in the presence of expansion predicate symbols.

5.4 Join Refinement by Indexing

Joining two tables of size $O(n)$ without any index is an expensive operation. A good indexing on tables reduces the join computation time dramatically, especially for the cases when the size of result table is of size the same order as of the input tables. We have implemented a simple*Hash Indexing* method based on the following function Hash, which assigns hash indices to entries of extended-hidden relations $T^{\overline{y_t}}_{\overline{x_t}}$ and $S^{\overline{y_s}}_{\overline{x_s}}$, invloved in a join, and is defined as below :

$\mathsf{Hash} : \{\overline{x_t} \cap \overline{x_s} \rightarrow A^k\} \rightarrow [1 \cdots \prod_{x_i \in \overline{x_S} \cap \overline{x_s}} DOM(x_i)]$

$\mathsf{Hash}(a_k, \ldots, a_1) = \sum_{i<k}(Image(a_i) * \prod_{j<k} DOM(x_j)) + 1$

where $k = |\overline{x_t} \cap \overline{x_S}|$, $DOM(x_i)$ is the size of the domain associated to x_i.
For the join of two relations $T^{\overline{y_t}}_{\overline{x_t}}$ and $S^{\overline{y_s}}_{\overline{x_s}}$, first a hash table is built for one of the relations, say $T^{\overline{y_t}}_{\overline{x_t}}$, that for each hash index keeps a linked list of tuples in $T^{\overline{y_t}}_{\overline{x_t}}$ associated to that index. Then the second relation, $S^{\overline{y_s}}_{\overline{x_s}}$, is scanned and for each tuple t with values $\overline{a}$ its associated hash index $\mathsf{Hash}(\overline{a})$ is computed. Tuples t' from $T^{\overline{y_t}}_{\overline{x_t}}$ that are kept by index $\mathsf{Hash}(\overline{a})$ in the hash table will be joined with t.

Choosing an effective indexing method requires study of different indexing methods and finding the one that fits to a particular application. Our hashing method significantly reduces the grounding time for problems we studied. The drawback is the size of hash tables. MXG can fail on join of some big tables due to the lack of memory space. Gnd-Hidden typically produces smaller tables than Gnd, so using Gnd-Hidden reduces this risk.

Chapter 6

Evaluation

Here we present an empirical comparison of the performance of MXG with with MidL, an independently developed solver for FO(ID)-MX, and with several well-known ASP solvers, on a collection of interesting benchmark problems. In the longer term, comparison with a wider range of approaches and systems, and over a wider variety of problems and instances, than we consider here is needed. Our current goal is to demonstrate that our approach can be used in practice. We choose comparison with ASP systems as they are the most similar competing approach which has reasonably mature theory and implementations. The results show that our MXG has performance comparable with the best known ASP solvers, and with MidL.

The following systems are compared in our experiments:

- MXG0.172 with MXC0.5, a SAT+Cardinality solver, denoted by 'MX'.
  MXG and MXC are available from http://www.cs.sfu.ca/research/groups/mxp/;
- MidL 2.2.0, a FO(ID) model expansion solver, with GidL 1.1.0 and Lparse 1.0.17 doing the grounding. GidL translates FO(ID) axioms to logical rules of Lparse and then runs Lparse to get ground rules, denoted by 'MidL'. GidL has Lparse1.0.17 embedded.
  MidL and GidL are available from http://www.cs.kuleuven.be/~dtai/krr/index.html;
- smodels 2.32, first ASP solver with Lparse 1.0.17 doing the grounding, denoted by 'smodels'.
  smodels and Lparse are available from http://www.tcs.hut.fi/Software/smodels;
- clasp 1.0.4, a ASP solver with clause learning with Lparse 1.0.17 doing grounding,

denoted by 'clasp'. clasp is available from http://www.cs.uni-potsdam.de/clasp/ and Lparse is available from http://www.tcs.hut.fi/Software/smodels;

- DLV 2006-7-14, an ASP solver for disjunctive logic programs. DLV has its own embedded grounder, denoted by 'dlv'.
  dlv is available from http://www.dbai.tuwien.ac.at/proj/dlv/.

The ASP community provides a collection of ASP axiomatizations and benchmark instances for a number of combinatorial problems in [1] These have been used, for example, in an ASP system competition [2] run in 2006/2007 [21]. We selected the following set of combinatorial problems from this collection:

- Graph k-Colouring
- Latin Square Completion
- Blocked Queens (BL-Queen)
- Social Golfer
- Bounded Spanning Tree (BST)

We ran the above systems on all instances of these problems provided on the Asparagus site [1]. We found most of the Asparagus graph colouring instances computationally trivial, so produced of colouring instances with few challenging instances from Asparagus plus a number of challenging instances from [3].

We present the MXG axiomatizations together with our computational results, organized by problem, later in this chapter. For evaluation of MXG+MXC, we used axiomatizations with cardinality constraints where this seems natural. Our MXG axiomatizations are not always the simplest possible, but neither are they highly refined to optimize performance.

We used axiomatizations for the other systems obtained as follows. For the ASP systems, we used the axiomatizations downloaded from Asparagus. For MidL, we used axiomatizations included in the MidL download package for K-colouring, Bounded Spanning Tree(BST) and Hamiltonian Cycle. To produce an Blocked N-Queens axiomatization, we extended the N-Queens axiomatization from the download package with an axiom enforcing the blocked cells constraint. For the remaining problems, we used a direct translation of our MXG axiomatizations, without cardinality constraints, into the MidL language, which differs slightly from the MXG language.

$problem$	$\#Instances$	MXG	$Lparse$
$Social - Golfer$	174	0.19(0.26)	**0.01**(0.00)
$BL - Queen - 28$	10	0.28(0.00)	**0.24**(0.00)
$BL - Queen - 48$	10	**1.47**(0.04)	1.92(0.02)
$BL - Queen - 50$	10	**1.62**(0.04)	2.23(0.04)
$BL - Queen - 56$	10	**2.38**(0.06)	3.48(0.03)
$Latin - Square$	100	**0.08**(0.00)	0.53(0.01)
$Colouring$	17	**1.11**(1.13)	1.28(1.66)
$BST - 35$	15	3.22(0.02)	**1.66**(0.06)
$BST - 45$	15	8.88(0.19)	**3.17**(0.07)

Table 6.1: Grounding time in seconds of MXG and Lparse

6.1 Comparing Grounding Times of MXG and Lparse

Our main interest is the overall performance of MXG as a solver, in comparison with other solvers. However, the overall running time of the solvers on an instance is composed of two parts – grounding time and ground solving time – and one may wonder if these components play significantly different roles in the performance of different systems.

Table 6.1 we compare grounding time of MXG and Lparse for the problems we have studied in this chapter. Each row in the table gives the mean and standard deviation (in parentheses) of the grounding times of MXG and Lparse for sets of instances studied in this chapter. (DLV, as distributed, does not support separation of grounding and solving times.) For the blocked queens and bounded minimum spanning tree problems, we partition instances into different sizes, while for others we consider the full set of instances as one distribution.

Comparing grounding times for grounders with different languages is somewhat of an apples-and-oranges comparison. For example, the syntax of MXG is much richer than that of Lparse, and thus the grounding problem could be harder. However, at least for the problems and instances studied here, grounding times of the two systems are not very different, and grounding times for both grounders are typically small in comparison with running times of the ground solvers.

In each row of Table 6.1, the smaller mean time is presented in bold face. We see that each is faster on about the same number of problems. In most cases, the grounding times for MXG and Lparse are not very different, and these differences are small in comparison with overall run-times (as reported in the following sections of this chapter).

For BST problem, the grounding time of MXG plays a large role in overall solving time of MXG + MXC. The difference of grounding time for Lparse and MXG for BST instances, is almost the whole difference of overall solving for clasp+Lparse, MXG + MXC. In all other cases, for both MXG and Lparse, grounding times are significantly smaller than ground solver times.

6.2 Overall Solving Performance

In the remaining sections of this chapter we compare all systems in terms of total solving time, which includes both grounding and ground solver time, and in the case of MXG includes "un-grounding" time also. In this section, we explain our method of comparison, including the conditions under which we run our experiments and the manner in which we present the resulting data.

6.2.1 Cumulative Performance Plots

When considering performance on NP-hard problems, looking at run-times for particular instances often is very interesting, but is often not the best way to see the overall trend in performance. Interpreting the table of running time for a large set of instances is not easy. Solvers of essentially the same quality often succeed on different instances for NP-hard problems, even within a collection of very similar instances.

Here, we present performance in a cumulative plot format that reduces emphasis on particular run-times, and gives a good overall picture of relative performance and *scaling* – how performance with problem size – of solvers on a collection of instances.

The performances plots (Figures 6.1 through 6.6) have the following format. The X-axis is logarithmic time in seconds; the Y-axis is the cumulative number of instances solved within a given time bound. For example, a point at $(10, 5)$ indicates that among the instances tested, 5 were solved in 10 seconds or less each, and the remaining all required more than 10 seconds each. We plot a point for each instance solved, so if the ith instance solved required n seconds, there is a point at (i, n). We connect the points for each solver with a curve purely as a visual aid. Notice that the X-axis of all plots is logarithmic to illustrate the important variations happening in small running times.

We ran all tests with a time cut-off of 30 minutes (1800 seconds), so the righthand edge of each plot is at 1800 seconds. The upper edge of each plot is at a value a bit larger than the number of instances in the relevant collection, so we can see the points when all instances were solved within the cut-off time. If (x,y) is the extreme upper-right point of the curve for solver A, then A solved y instances in x seconds or less, and failed to solve any of the remaining instances within the 30-minute cut-off. In all cases, we report the total time for both grounding and solving. For instances which are not computationally trivial, Lparse and MXG grounding times are almost always a small fraction of solving time.

6.2.2 Test Platform

All tests were run on Sun Fire VZ20 Dual Opteron computers with two 2.4 GHz AMD Opteron 250 processors having 1MB cache and 2GB of RAM per processor. The machines were running Suse Enterprise Linux 2.6.11. The executables for DLV, clasp, MidL, MXC and MXG were downloaded from the respective solver sites. Executables for smodels was compiled with gcc version 3.3.4, using the default settings of the makefile provided with the solver source from http://www.tcs.hut.fi/Software/smodels.

In Sections 6.3 to 6.7, we present the plots showing the relative performance of the various systems on the chosen benchmark problems, and also the MXG axiomatizations we used. We verified every solution produced during our tests with both MXG and smodels.

For each problem, we estimate an order of preference of solvers based on performance. Since we prefer to reward good scaling over fast solving of small instances, our ordering is as follows: we prefer those solvers that solved the most instances within the cut-off time, and among those we prefer the one that minimized the maximum time for solving an instance. Solver order could change with an increased cut-off time, but this will be the case with any preference scheme that rewards good scaling.

6.3 Graph K-Colouring

Graph colouring is a classic and well-studied NP-hard search problem. An instance consists of a graph and a number K, and a solution is a proper K-colouring of the graph. That is, we want a function mapping vertices to a set of K colours, so that no two adjacent vertices are mapped to the same colour.

We take our instance to consist of the graph plus the set of colours. So we have two sorts: Vtx and Clr. The axioms simply says there is a binary relation Colour which must be a proper colouring of the vertices:

```
Given:
        type Vtx Clr;
        Edge(Vtx, Vtx)
Find:
        Colour(Vtx, Clr)
Satisfying:
        Colour(MIN,MIN)                                                  (1)
        ∀ x y z < y :¬(Colour(x, y) ∧ Colour(x, z))                      (2)
        ∀ x y : (Edge(x,y) ⊃ (∀ z : ¬(Colour(x, z) & Colour(y,z))))      (3)
        ∀ x: ∃ y : Colour(x, y)                                          (4)
```

Our test set consisted of 17 instances, 5 3-colouring instances from Asparagus and 12 instances from the LEI category of the graph colouring benchmark collection at [3]. These latter are are challenging instances on graphs of 450 vertices, variously with 5, 15 and 25 colours. All 17 instances are colourable with the alloted number of colours. The performance of the solvers is shown in Figure 6.1. The figure clearly shows the following order of solvers, from best to worst: MXG+MXC, clasp, MidL, DLV, smodels. Notice that no solver was successful on all instances: MXG+MXC and clasp solved 13 of the 17 instances, while others solved 10 or fewer. The four instances that were not solved by MXG went unsolved by all solvers tested.

6.4 Latin Square Completion

A Latin Square (or Quasigroup) is an n by n matrix with elements in $\{1, \ldots, n\}$, where every row and every column has every possible element. In the Latin Square Completion problem, an instance is the set $\{1, \ldots, n\}$ plus prescribed values for certain elements. The task is to construct a Latin square consistent with the prescribed elements. Existence of such a completion is NP-complete.

In our MXG axiomatization, Cell denotes matrix elements. The bijection of each row, column on $\{1, \ldots, n\}$ is expressed by cardinality constraints (2), (3). Axiom (1) imposed the preassigned elements. Axiom (4) ensures that each cell takes exactly one element.

Given :

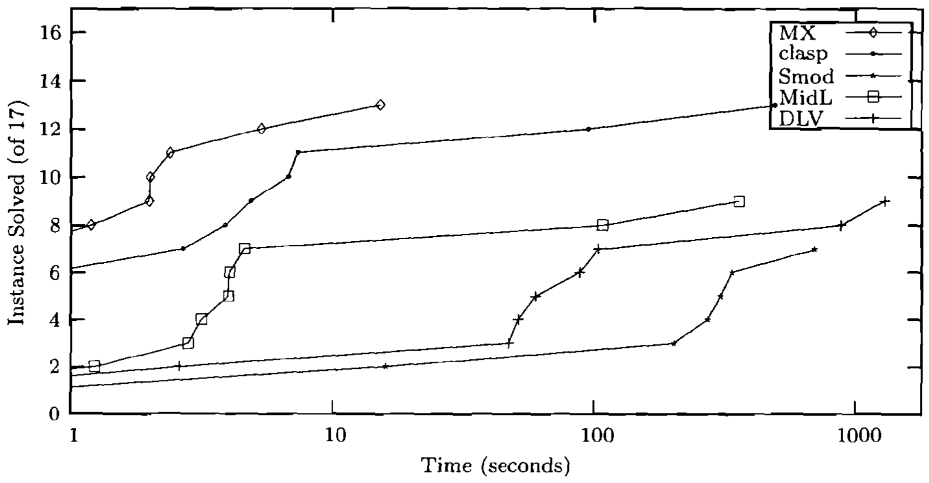

Figure 6.1: Performance on Graph K-Colouring

```
        type Num;
        Preassigned(Num, Num, Num)
Find :
        Cell(Num, Num, Num)
Satisfying:
```

$\forall$ x y z : (Preassigned(x, y, z) $\supset$ Cell(x, y, z)) (1)

$\forall$ x y : CARD(1, z; Cell(x, y, z)) (2)

$\forall$ x z : CARD(1, y; Cell(x, y, z)) (3)

$\forall$ z y : CARD(1, x; Cell(x, y, z)) (4)

The test set consists of 100 instances from Asparagus. All are of size 30-by-30, and all have solutions. The apparent ordering is: MXG, clasp, MidL, smodels, and DLV. The performance plot is shown in Figure 6.2. MXG sees all the instances as essentially identical and trivial.

To demonstrate the effect of cardinality constraints on the overall performance of solvers, in Figure 6.3 we compare the performance of MXG for our best axiomatization without cardinality constraints, denoted by MX-Norm, with the timing of MXG for axiomatization with cardinality, and clasp. The results show that clasp beat MXG when we do not have cardinality constraints.

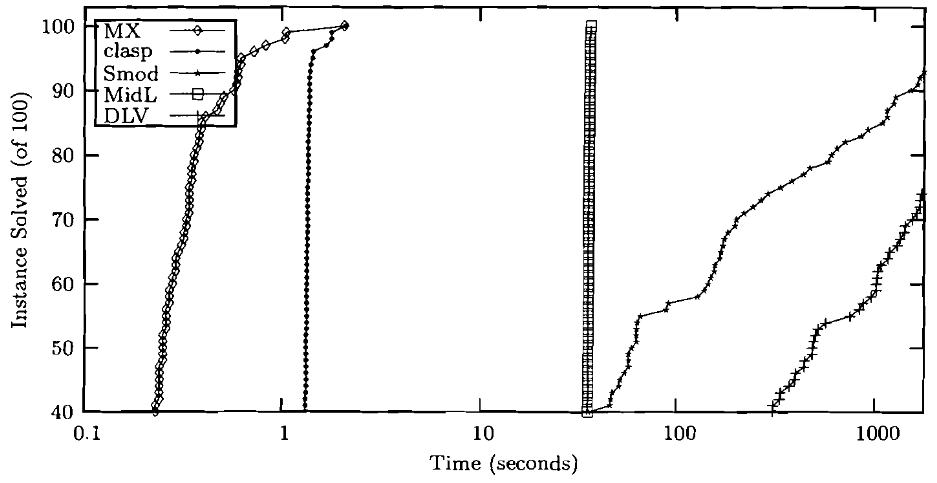

Figure 6.2: Performance on Latin Square

6.5 Blocked Queens

The n-Queens problem is to place n Queens on an $n \times n$ chessboard, so that no two attack each other. In the blocked queens problem, we are not allowed to put a queen on certain cells. Existence of such a completion is NP-complete.

Current version of MXG has no arithmetic. We defined predicate Diff, which represents subtraction (Diff(i, x, y) holds iff y - x = i + 1), inductively with a Horn-ID(see Section 2.4.3). Axioms 6 and 7 ensure that one queen is located in each row and column. Axioms 3 and 4 check that at most one queen is on each diagonal.

```
Given :
        type Num;
        Block(Num, Num)
Find :
        Queen(Num, Num)
Satisfying:
        Diff(Num,Num,Num)
        {Diff(i,x2,y2) ← (x2 = MIN ∧ y2 = i)                                    (1)
        Diff(i,x2,y2) ← (Diff(i,x1,y1) ∧ SUCC(x1,x2) ∧ SUCC(y1,y2))}           (2)
        ∀ x y: (Queen(x,y) ⊃ ¬Block(x,y))                                       (3)
        ∀ i x1 y1 x2 y2: (y1<y2) ⊃ ((x1<x2) ⊃
                ¬(Queen(x1,y1) ∧ Queen(x2,y2) ∧ Diff(i,x1,x2) ∧ Diff(i,y1,y2)))  (4)
        ∀ i x1 y1 x2 y2: (y2<y1) ⊃ ((x1<x2) ⊃
                ¬(Queen(x1,y1) ∧ Queen(x2,y2) ∧ Diff(i,x1,x2) ∧ Diff(i,y2,y1)))  (5)
```

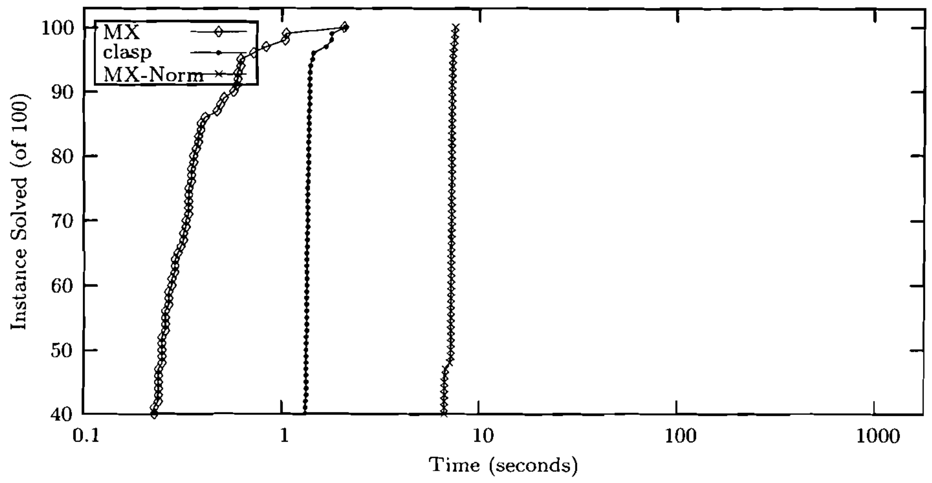

Figure 6.3: Performance of MXG on Latin Square With/Without Cardinality Constraints

$$\forall x : \mathrm{CARD}(1;y;\mathrm{Queen}(x,y)) \quad (6)$$
$$\forall y : \mathrm{CARD}(1;x;\mathrm{Queen}(x,y)) \quad (7)$$

The test set consists of 40 instances from Asparagus, of sizes from 28*28 to 56*56, 20 of which have solutions. While having a similar flavor to Latin Square completion, the performance profile (at least on the Asparagus instances) is different, in that all solvers found the instances to vary considerably in difficulty. The order has changed to: clasp, MXG, smodels, MidL, and DLV. (Figure 6.4)

6.6 Social Golfer

The goal is to schedule $g \times s$ golfers into g groups of s players over w rounds (weeks), such that no two golfers play in the same group more than once. This problem has a lot of symmetric solutions [6] [5]. For example a valid scheduling for weeks $1, \ldots, w$ is also a valid scheduling for any $w!$ combinations of weeks. If the problem does not have a solution it takes a lot of time from SAT solver to verify all these isomorphic solutions. To remove these symmetries from problem solutions, a well known technique is to add some extra constraints to break symmetries [51] [5]. In our MXG axiomatization, axioms 5, 6-1 to 6-4, and 7-1 to 7-4 are extra and added for breaking symmetries of players, groups, and weeks respectively.

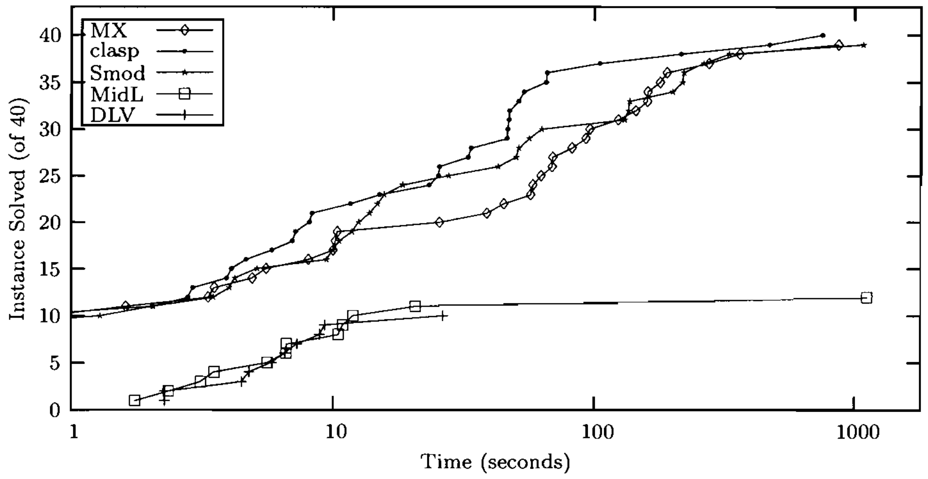

Figure 6.4: Performance on Blocked Queens

Plays(p_i, w_i, g_i) assigns player p_i to play in group g_i at week w_i. If two players $p_1 < p_2$ play in the same group at week w, they are socialized: *Soc*(w, p_1, p_2) (Axiom 1). Axiom (2) states exactly Groupsize number players play in the same group in each week. Each player should play in exactly one group each group is checked by axiom (3), and two players can not be socialized more than once by axiom (4).

```
Given :
        type Players Groups Weeks;
        Groupsize : Players
Find :
        Plays(Players, Weeks, Groups)
Satisfying:
        MP(Weeks, Groups, Players)
        SP(Weeks, Players)
        Soc(Weeks, Players, Players)
        ∀ pl p2>pl w : (Soc(w,pl,p2) ⇔ (∃ g : (Plays(pl,w,g) ∧ Plays(p2,w,g))))    (1)
        ∀ w g: CARD(Groupsize; p; Plays(p,w,g))                                   (2)
        ∀ p w : CARD(1; g; Plays(p, w, g))                                        (3)
        ∀ pl p2>pl : UB(1; w; Soc(w,pl,p2))                                       (4)
        ∀ gl g2>gl pl p2<pl : ¬(Plays(pl, MIN, gl) ∧ Plays(p2, MIN, g2))          (5)
        ∀ w p : (SP(w,p) ⊃ (p > MIN ∧ Plays(p, w, MIN)))                          (6-1)
        ∀ w p2>MIN pl>p2 : ¬(SP(w,pl) ∧ Plays(p2, w, MIN))                        (6-2)
        ∀ w : CARD(1; p; SP(w,p))                                                 (6-3)
        ∀ wl w2>wl pl p2≤pl : ¬(SP(wl, pl) ∧ SP(w2, p2))                          (6-4)
        ∀ w : Plays(MIN, w, MIN)                                                  (7-1)
```

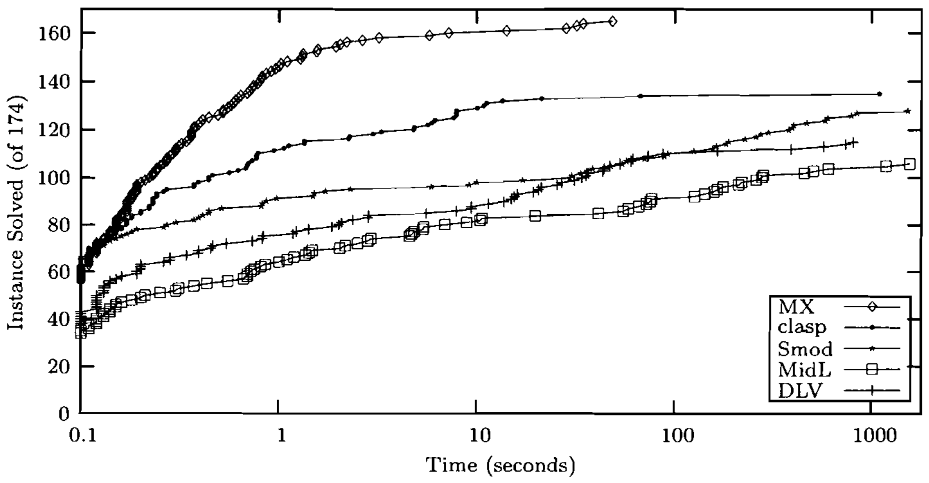

Figure 6.5: Performance on Social Golfer

$\forall$ w g : CARD(1; p; MP(w,g,p))	(7-2)
$\forall$ w g p : (MP(w,g,p) $\supset$ (! pl < p : $\neg$(Soc(w, pl, p))))	(7-3)
$\forall$ w pl p2$\leq$pl gl g2>gl : $\neg$(MP(w, gl, pl) $\wedge$ MP(w, g2, p2))	(7-4)

The test set consists of 174 instances from Asparagus, spanning(but not covering) the parameter range: number of weeks from 2 to 8; group size from 2 to 6; number of groups from 2 to 8. We know 72 instances to have solutions and 64 to have no solution, leaving 38 of unknown status. The order of solvers is: clasp, MXG, smodels, DLV, Cmodels, MidL.

6.7 Bounded Spanning Tree

A spanning tree of a graph is a sub-graph that is a tree and visits every vertex. A spanning tree can be found in linear time. In this version of the problem, an instance consists of a directed graph and a bound *Bound*, and we require a directed spanning tree in which no vertex has out-degree larger than *Bound*. Existence is NP-complete.

Axioms (3),(4),(5) respectively state that Bstedge edges are subset of graph edges, each vertex has in-degree one, and out-degree at most Bound. To make sure Bstedge does not have any loops, we define a mapping over vertices of graph (axioms 1, 2), and force the vertices of the spanning tree should be ordered according to mapping(axioms 6,7).

Given :

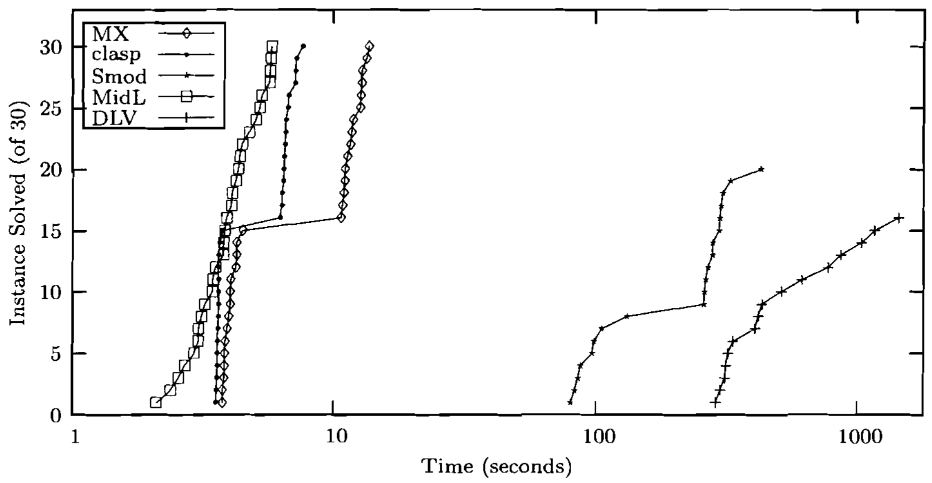

Figure 6.6: Performance on Bounded Spanning Tree

```
          type Vtx;
          Edge(Vtx,Vtx);
          Bound : Vtx
Find :
          Bstedge(Vtx,Vtx)
Satisfying:
          Map(Vtx,Vtx)
          ∀ x: CARD(1, y; Map(x,y))                                       (1)
          ∀ y: CARD(1, x; Map(x,y))                                       (2)
          ∀ v u : (Bstedge(u,v) ⊃ Edge(u,v))                              (3)
          ∀ x : UB(1, y; Bstedge(y,x))                                    (4)
          ∀ u : UB(Bound, v; Bstedge(u, v))                               (5)
          ∀ u v x y≤x : ¬(Map(x,u) ∧ Map(y,v) ∧ Bstedge(u,v))             (6)
          ∀ v f>MIN: (Map(v, f) ⊃ (∃u:Bstedge(u,v)))                      (7)
```

The test set is 30 instances from Asparagus, some with 35 and some with 45 vertices. All have solutions. The solver order is: MidL, clasp, MXG, smodels, and DLV. The difference between MidL and MXG+MXC is entirely accounted for by grounding time.

Chapter 7

Related Work

In this chapter, we survey some other declarative constraint programming systems for representation and solving NP-search problems which, like MXG, have as a goal to represent problems in a high-level language and reduce the high-level specification into a low-level one, for which they have a solver that can find solutions. We will also outline the grounding methods used in these systems.

7.1 Other FO(ID)-MX Solvers

There are only two solvers, other than MXG, for FO(ID)-MX: IDSAT [50], and MidL [41].

IDSAT is an earlier solver for FO(ID)-MX produced at SFU. It used a slightly modified version of Lparse [53] as a front-end. The problem specification was in input syntax of Lparse, but with FO(ID) semantics. The ground rules generated by Lparse were translated to PC(ID), the propositional analog of FO(ID)), producing a formula consisting of a set of propositional clauses with a set of propositional inductive definitions. To use a general SAT solver for finding models, a level-mapping reduction for the from the PC(ID) inductive definitions to SAT was performed. While performance was not unacceptable on benchmark problems considered, the level-mapping reduction was not *faithful*. That is, it did not preserve a one-to-one correspondence between solutions and satisfying assignments of the formula, which could be important for some purposes. Unfortunately, faithful reductions are more difficult to produce, and are often larger, so their use may reduce solver performance.

MidL is a solver for FO(ID)-MX developed by Marc Denecker's group at KU Leuven [41]. The MidL input language is similar to that of MXG. Some differences, at the time of writing, are that the MidL language does not have ordered domains, and cardinality constraints, but does have a form of arithmetic, and supports unrestricted inductive definitions. The MidL grounder is based on Lparse. Inputs to MidL are translated to the Lparse language, and the ground rules produced by Lparse are then transformed to a PC(ID) theory. The MidL solver engine handles PC(ID) logic natively, rather than using reduction to SAT, as described in [39].

7.2 ASP Systems

Answer Set programming (ASP) [38][48] is a declarative programming approach based on the language of logic programs with stable model semantics [23]. Problem constraints are described by rules similar to function-free Prolog rules, and the solver searches for the minimal Herbrand models of the set of rules. An input program is normal ($\vee$-free) if the head of each rule is a single predicate, and is a disjunctive logic program if head of some rule is a disjunction. All ASP systems we know of operate by grounding to a ground logic program and then application of a ground solver for this language. While there is quite a variety of ASP solvers, Smodels [54], Cmodels [35], ASSAT [37], Cmodels-2 [36], DLV [33], Clasp [20], there are merely three major ASP grounders, Lparse [53], DLV's grounding component [33], and GrinGo [22]. Lparse [53] was originally developed as the front-end for the ground solver Smodels [54], but has since been used as the grounder for many other ground ASP solvers. Its language extends normal logic programs with weight constraints (a generalization of cardinality constraints), and arithmetic. Gringo [22] is an Lparse alternative, recently released by developers of Clasp [20]. The current version of Gringo supports a large fragment of the Lparse language. DLV [33] has it own grounder, embedded with solver, and is the state-of-the-art solver for disjunctive logic programs.

All three grounders use a similar approach for generating ground rules: a procedure "instantiate" recursively substitutes constant symbols for variables of a rule to produce ground instances of rules. If the value of a ground body predicate is known at the substitution time, then it is either eliminated from the ground rule (if it is true) or the rule is deleted (if it is false). A simple "instantiate" procedure backtracks when all variables of a rule are instantiated, or an assignment of variables makes a rule invalid to set the next variable assignment. Backtracking generates all rule instances, but it is possible to get a lot of redundant and useless ground instances of rules. 'Backjumping' is a technique used in the instantiation modules of DLV [32] and GrinGo [22] to avoid generation of these. The backjumping algorithm of Gringo reduces the number of redundant rules more than that of DLV.

For ASP solvers, instances are provided in the form of a set of ground atoms, which formally are part of the same logic program as the problem specification. Separation of problem and instance descriptions is considered important [38], but maintained only as a convention which is not always followed in practice. The semantics is based on Herbrand models, which entails significant restrictions on the use of function symbols. In MXG, no such mechanism is required to invoke closure on the universe, which is given explicitly with the instance. Thus, functions can be added to MXG more easily than to the ASP languages.

The built-in recursion in ASP is often convenient, particularly for axiomatizing problems involving sequences of events, such as verification and planning problems. One problem, in our view, is that the entire formula (logic program) is involved in the recursion. Many properties which can be easily and naturally expressed classically require a less natural expression within this recursion.

7.3 Constraint Modelling Languages

Several languages known as "constraint modelling" or "constraint specification" languages provide declarative languages for describing search (and decision and optimization) problems. Examples include ESRA [18], Essence [19] and Zinc [12]. These languages are not explicitly logic-based, although it is possible to view their specifications as MX specifications for a suitable logic. Implementations of these languages work by either translation to a high-level programming languages (including Constraint Logic Programming (CLP) languages) or by a grounding process where the ground language is for some "CSP solver" – roughly, a solver for some generalization of SAT to non-boolean domains.

7.4 Finite Model Finders (Generators)

A finite model finder is a tool trying to find models for a (sorted or unsorted) FO theory. Searching for finite models for FO theories is complementary to theorem proving. Theorem provers use the finite model finders to prove the consistency of a FO theory (find a model) or to disprove its validity (by finding a counter-example). Methods of finding finite models can mostly be categorized in two different styles [9]: MACE-style, named after the tool MACE [43], and SEM-style, named after the tool SEM [57]. SEM-style methods perform backtracking search on the problems directly, while MACE-style methods transform the problem of finding a model in a finite domain to a SAT problem. MACE-style tools, either run a SAT solver on the SAT problem to find the model, as in ModGen [30], or they have some satisfiability solving methods built-in that search for models, as in MACE [43].

Finite model finders can search for a model with fixed domain, or start with a domain of basic size and goes through consecutive stages of increasing the domain sizes up to a certain value (specified in the problem definition).

There are different input languages for model finders: Problems can be expressed by quantifier-free FO clauses, where all the variables are implicitly quantified universally, and existentially quantified variables are replaced by some functions (*Skolem* functions). Some systems accept arbitrary FO sentences, but transform FO sentences to quantifier-free FO clauses, introducing *Skolem* functions where needed to replace the existential quantifiers. Most model generators allow the user to give some certain properties, such as quasigroup, bijection, equality, order, etc, to vocabulary symbols easily. These properties usually provide some means to express isomorphisms in problem models.

SAT-based finite model finders have to transform FO clauses to propositional clauses. Transformation is by "flattening" the clause, replacing complex predicate arguments (function symbols) by new variables, and then "instantiating" the clause, by applying all substitutions of domain elements for variables, and evaluating out the known literals to generate a set of propositional clauses. During flattening new predicates are needed to be defined to state the relation of new variable with variables of a replaced function. For each predicate defined for a function, a set of clauses is added to problem theory to express "image uniqueness", and "totality" of the new predicates.

SAT-based model finders that look for a model in a fixed multi-sorted domain, such as ModGen [30], have quite similar solving method as MXG. Their main differences are in the underlying logic of the modeling languages, and the method of grounding. MXG augments FO logic, with inductive definitions, cardinality constraints, and orders, to provide a more practical modelling language than "bare-bones" classical logic. Finite model finders introduce new predicates in the "flattening" step, with a set of clauses for expressing "image uniqueness" and "totality" of these predicates. This increases the size of clauses significantly for large domains. MXG's grounding algorithm applies relational algebra operations on extended relations to compute answer to each formula. While for finite model finders an "instantiate" procedure substitutes all variables with domain elements, in a backtrack procedure.

Chapter 8

Conclusion and Future work

NP-hard search problems arise frequently in application areas ranging from software verification to bio-informatics, and often the inability to solve sufficiently large instances within practical time bounds is a significant obstacle to practical work. Practitioners in these areas need effective tools, but producing good implementations requires considerable effort; moreover, there are relatively few techniques which work well in practice, and these are implemented over and over in different domains. There appears to be a role for well-implemented general-purpose tools for modelling and solving search problems. Such tools could greatly facilitate the work of many practitioners who now must get by with poorly adapted tools and techniques, or invest a great deal of energy implementing domain-specific search algorithms.

To demonstrate that model expansion framework can be the basis for such technology, we have implemented a solver, MXG, for parameterized FO(ID+Card) model expansion. The performance of the solver clearly shows the feasibility of the approach, providing performance competitive to more mature ASP systems. The solving method in MXG is to reduce the FO(ID+Card) model expansion problem to a SAT problem, possibly including cardinality clauses, and then run a SAT(+Card) solver. Reduction is a done by a novel polynomial time grounding algorithm, based on applying relational algebra operation to extended-hidden relations. Although MXG language is less restricted than language of Lparse, and grounding should be harder, the grounding time performance of MXG is comparable by that of Lparse for problem and instances we studied.

8.1 Future Work

MXG is in its early stages of development, but already performs quite well. Future work includes:

- handling general inductive definitions,
- using cardinality constraints in arbitrary FO formula, and adding more aggregate,
- providing arithmetic built-in in MXG,
- using a more effective hashing function to index tables, for join and other relational algebra operations.

Appendix A

MXG Problem Specification Grammar

```
<TheoryFile>        ::= <GivenPart> <FindPart> <SatisfyingPart>
<GivenPart>         ::= Given : <TypeDCL> <SymbDCL>
<TypeDCL>           ::= type <TypeNames> ;
<TypeNames>         ::= <TypeName> | <TypeNames> <TypeName>
<SymbDCL>           ::= | <SymbDCL> <a_SymbDCL>
<a_SymbDCL>         ::= <PredDCL> | <ConstDCL>
<PredDCL>           ::= <PredName> ( <PredTypes> )
<PredTypes>         ::= <TypeName> | <PredTypes> , <TypeName>
<ConstDCL>          ::= <ConstName> : <TypeName>
<FindPart>          ::= Find : <SymbDCL>
<SatisfyingPart>    ::= Satisfying : <SatisfyingRules>
<SatisfyingRules>   ::= <a_SatisfyingRule> |
                        <SatisfyingRules> <a_SatisfyingRule>
<a_SatisfyingRule>  ::= <an_Axiom> | <a_SymbDCL>
<an_Axiom>          ::= <FO_Formula> | { <IDD_Rules> }
<FO_Formula>        ::= <Unitary_Formula> |
                        <FO_Formula> <Connective> <Unitary_Formula>
<unitary_formula>   ::= ( <FO_Formula> ) | <QuantPart> : <Unitary_Formula> |
                        ~<Unitary_Formula> | <atomic_formula> |
       <Card_formula>
<atomic_formula>    ::= <PredName> ( <Args> ) |
                        SUCC ( <Args> ) |
                        <Ord_Relation>
<Card_formula>      ::= <Card_Symb> (<Bound>; <Variables>; <fof_formula>)
<Ord_Relation>      ::= <VarName> <OrdOperator> <VarName> |
                        <VarName> <OrdOperator> <ConstName> |
                        <VarName> <OrdOperator> MIN |
                        <VarName> <OrdOperator> MAX |
                        <ConstName> <OrdOperator> <VarName> |
                        MIN <OrdOperator> <VarName> |
                        MAX <OrdOperator> <VarName>
<QuantPart>         ::= <Quantifier> <Variables>
<Quantifier>        ::= ? | !
<Variables>         ::= <a_Var> | <Variables> <a_Var>
<a_Var>             ::= <VarName> | <Ord_Relation>
<Args>              ::= <an_Arg> | <Args> , <an_Arg>
<an_Arg>            ::= <VarName> | MIN | MAX | <ConstName>
<OrdOperator>       ::= < | <= | > | >= | =
<Connective>        ::= & | <vline> | => | <=>
<vline>             ::= |
<IDD_Rules>         ::= <IDD_Rule> | <IDD_Rules> <IDD_Rule>
<IDD_Rule>          ::= <PredName> ( <Args> ) <- <QF_FO_Formula>
<QF_FO_Formula>     ::= <atomic_formula> | ( <QF_FO_Formula> ) |
                        <QF_FO_Formula> <BinaryOperator> <atomic_formula> |
                        ~( <QF_FO_Formula> ) | ~<atomic_Formula>
```

Appendix B

MXG Instance Description Grammar

```
<InstanceFile>     ::= <an_InstancePart> | <InstanceFile> <an_InstancePart>
<an_InstancePart> ::=  <InstnaceType> | <InstnacePred> | <InstanceConst>
<InstnaceType>     ::= <TypeName> = [ <TypeElements> ]
<TypeElements>     ::= <Character>..<Character> |
                       <Number>..<Number> | <TypeElementSet>
<TypeElementSet>   ::= <a_TypeElement> |
                       <TypeElementSet> ; <a_TypeElement>
<a_TypeElement>    ::= <ElementName> | <Number> | Character
<InstnacePred>     ::= <PredName> = { <Tuples> }
<Tuples>           ::= <a_Tuple> | <Tuples> ; <a_Tuple>
<a_Tuple>          ::= <a_TypeElement> | <a_Tuple> , <a_TypeElement>
<InstanceConst>    ::= <ConstName> = <a_TypeElement>
```

Bibliography

[1] Asparagus repository, http://asparagus.cs.uni-potsdam.de/.

[2] The first answer set programming system competition 2007, http://asparagus.cs.uni-potsdam.de/contest/.

[3] http://mat.gsia.cmu.edu/color/instances.

[4] Model expansion project website, http://www.cs.sfu.ca/research/groups/mxp.

[5] Francisco Azevedo and Hau Nguyen Van. Symmetry breaking and extra constraints for the Social Golfers problem, 2007.

[6] Francisco Azevedol and Hau Nguyen Van. Extra constraints for the Social Golfers problem, 2006.

[7] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.

[8] Alonzo Church. A note on the entscheidungsproblem. *Symbolic Logic*, 1(1):40–41, 1936.

[9] K. Claessen and N. orensson. New techniques that improve MACE-style finite model finding, 2003.

[10] Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977.

[11] Stephen A. Cook. The complexity of theorem proving procedures. In *Annual ACM Symp. on Theory of Computing*, pages 151–158, 1971.

[12] Maria J. García de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace. The modelling language Zinc. In *CP*, pages 700–705, 2006.

[13] M. Denecker and E. Ternovska. A logic of non-monotone inductive definitions and its modularity properties. *Lecture Notes in Computer Science*, 2923:47–60, 2004.

[14] M. Denecker and E. Ternovska. A logic of non-monotone inductive denitions. *ACM Transactions on Computational Logic*, 2008.

[15] Marc Denecker. Extending classical logic with inductive definitions. *Lecture Notes in Computer Science*, 1861:703–717, 2000.

[16] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *J. Log. Program.*, 1(3):267–284, 1984.

[17] Ronald Fagin. Generalized First-Order spectra and polynomial-time recognizable sets. *Complexity of Computation*, 7:43–73, 1974.

[18] Pierre Flener, Justin Pearson, and Magnus Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *CP*, page 971, 2003.

[19] Alan M. Frisch, Matthew Grum, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The design of ESSENCE: A constraint language for specifying combinatorial problems. In *IJCAI*, pages 80–87, 2007.

[20] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *clasp* : A Conflict-Driven answer set solver. In *LPNMR*, pages 260–265, 2007.

[21] Martin Gebser, Lengning Liu, Gayathri Namasivayam, André Neumann, Torsten Schaub, and Miroslaw Truszczynski. The first answer set programming system competition. In *LPNMR*, pages 3–17, 2007.

[22] Martin Gebser, Torsten Schaub, and Sven Thiele. GrinGo : A new grounder for answer set programming. In *LPNMR*, pages 266–271, 2007.

[23] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[24] Dimopoulos Y. Haslum P. Gerevini, A. and A. Saetti. 5th international planning competition (ipc-5), web page: http://zeus.ing.unibs.it/ipc-5/, 2006.

[25] Belaid Benhamou Gilles Audemard. Reasoning by symmetry and function ordering in finite model generation. In *CADE*, pages 226–240, 2002.

[26] W. Harvey. Symmetry breaking and the Social Golfer problem. In *SymCon-01: Symmetry in Constraints*, pages 9–16, 2001.

[27] Lawrence J. Henschen and Larry Wos. Unit refutations and Horn sets. *J. ACM*, 21(4):590–605, 1974.

[28] Neil Immerman. *Descriptive Complexity*. Springer, 1999.

[29] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

[30] Sun Kim and Hantao Zhang. ModGen: Theorem proving by model generation. In *National Conference on Artificial Intelligence*, pages 162–167, 1994.

[31] Nicola Leone, Simona Perri, and Francesco Scarcello. Improving ASP instantiators by join-ordering methods. In *LPNMR*, pages 280–294, 2001.

[32] Nicola Leone, Simona Perri, and Francesco Scarcello. Backjumping techniques for rules instantiation in the DLV system. In *NMR*, pages 258–266, 2004.

[33] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM TOCL*, 7(3):499–562, 2006.

[34] Leonid Libkin. *Elements of Finite Model Theory.* Springer, 2004.

[35] Yuliya Lierler. cmodels - SAT-based disjunctive answer set solver. In *LPNMR*, pages 447–451, 2005.

[36] Yuliya Lierler and Marco Maratea. Cmodels-2: SAT-based Answer Set Solver enhanced to non-tight programs. In *LPNMR*, pages 346–350, 2004.

[37] Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artif. Intell.*, 157(1-2):115–137, 2004.

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

[39] Maarte Marien, Rudradeb Mitra, Marc Denecker, and Maurice Bruynooghe. Satisfiability checking for PC(ID). In *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR*, volume 3835 of *Lecture Notes in Computer Science*, pages 565–579, 2005.

[40] Maarten Mariën, Rudradeb Mitra, Marc Denecker, and Maurice Bruynooghe. Satisfiability checking for PC(ID). In *LPAR*, pages 565–579, 2005.

[41] Maarten Marien, Johan Wittocx, and Marc Denecker. The IDP framework for declarative problem solving. In *Search and Logic: Answer Set Programming and SAT*, pages 19–34, 2006.

[42] Maarten Mariën, Johan Wittocx, and Marc Denecker. The IDP framework for declartive problem solving. In *Search and Logic: Answer Set Programming and SAT*, pages 19–34, 2006.

[43] William McCune. A Davis-Putnam program and its application to finite first-order model search: quasigroup existence problems. Technical report, Argonne National Laboratory, 1994.

[44] David Mitchell and Eugenia 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.

[45] David Mitchell, Eugenia Ternovska, Faraz Hach, and Raheleh Mohebali. Model expansion as a framework for modelling and solving search problems. Technical Report TR 2006-24, School of Computing Science, Simon Fraser University, December 2006.

[46] David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In *Proc., Twentieth National Conf. on Artificial Intelligence (AAAI-05)*, pages 430–435, July 2005.

[47] Raheleh Mohebali, Faraz Hach, and David G. Mitchell. MXG: A model expansion grounder and solver. In *LPAR*, 2007.

[48] I. Niemela. Logic programs with stable model semantics as a constraint programming paradigm. In I. Niemela and T. Schaub, editors, *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pages 72–79, 1998.

[49] Murray Patterson, Yongmei Liu, Eugenia Ternovska, and Arvind Gupta. Grounding for model expansion in k-guarded formulas with inductive definitions. In *IJCAI*, pages 161–166, 2007.

[50] Nikolay Pelov and Eugenia Ternovska. Reducing inductive definitions to propositional satisfiability. In *ICLP*, pages 221–234, 2005.

[51] B. Smith. Reducing symmetry in a combinatorial design problem, 2001. RR 01, University of Leeds (UK), School of Computer Studies, 2001.

[52] L.J. Stockmeyer. The complexity of decision problems in automata theory, 1974.

[53] Tommi Syrjänen. Lparse 1.0 user's manual.

[54] Tommi Syrjänen and Ilkka Niemelä. The Smodels system. In *6th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 434–438, 2001.

[55] G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 466–483. Springer, Berlin, Heidelberg, 1983.

[56] Moshe Y. Vardi. The complexity of relational query languages. In *In Proceedings of the 14th ACM Symposium on Theory of Computing*, pages 137–146, 1982.

[57] Jian Zhang and Hantao Zhang. System description: Generating models by SEM. In *CADE*, pages 308–312, 1996.