SOLVING MODEL EXPANSION TASKS: SYSTEM DESIGN AND MODULARITY

by

Xiongnan (Newman) WU

B.Sc., Simon Fraser University & Zhejiang University, 2010

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

© Xiongnan (Newman) WU 2012
SIMON FRASER UNIVERSITY
Summer 2012

All rights reserved. However, in accordance with the Copyright Act of Canada, this work may be reproduced without authorization under the conditions for Fair Dealing. Therefore, limited reproduction of this work for the purposes of private study, research, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.

APPROVAL

Name: Xiongnan (Newman) WU

Degree: Master of Science

Title of Thesis: Solving Model Expansion Tasks: System Design and Modularity

Examining Committee: Dr. Greg Mori, Associate Professor, Computing Science
Simon Fraser University
Chair

Dr. Evgenia Ternovska, Associate Professor, Computing Science
Simon Fraser University
Senior Supervisor

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

Dr. James P. Delgrande, Professor, Computing Science
Simon Fraser University
Examiner

Date Approved: August 7, 2012

Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the "Institutional Repository" link of the SFU Library website (www.lib.sfu.ca) at http://summit/sfu.ca and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

While licensing SFU to permit the above uses, the author retains copyright in the thesis, project or extended essays, including the right to change the work for subsequent purposes, including editing and publishing the work in whole or in part, and licensing other parties, as the author may desire.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, British Columbia, Canada

revised Fall 2011

Abstract

In this thesis, we present the Enfragmo system for representing and solving combinatorial search problems. The system supports natural specification of problems by providing users with a rich language, based on an extension of first order logic. Since the specification language is high level, Enfragmo provides combinatorial problem-solving capability to users without expertise in advanced solver technology. On the other hand, some search problems, e.g., the task of constructing a logistics service provider relying on local service providers, are inherently modular. The framework is extended to represent a modular system. It allows one to combine modules on an abstract model-theoretic level, independently from what languages are used for describing them. In this thesis, an algorithm for finding solutions to such modular systems is proposed. We show that our algorithm closely corresponds to what is done in practice in different areas such as Satisfiability Modulo Theories, Integer Linear Programming, and Answer Set Programming.

Keywords: declarative programming; search problems; model expansion; modularity

To my wife Yidan.

"There are two great days in a person's life -
the day we are born
and the day we discover why."
- William Barclay

Acknowledgments

I would like to thank everyone who has helped me during my research at Simon Fraser University. First, I would like to thank Dr. Eugenia Ternovska, my senior supervisor, for her support and guidance on my research. Next, I would also like to thank my colleagues, Amir Aavani and Shahab Tasharrofi for the collaborative work on the research project. Finally, I would like to express my gratitude to Dr. David G. Mitchell and Dr. James P. Delgrande for the valuable discussions and feedbacks.

Contents

Approval ii

Abstract iii

Dedication iv

Quotation v

Acknowledgments vi

Contents vii

List of Tables x

List of Figures xi

1 Introduction 1

2 Background 3

2.1 Model Expansion 3

3 The Enfragmo System 5

3.1 Introduction 5

3.1.1 My Contributions 6

3.2 Preliminaries 7

3.2.1 Ordered Structures and Multi-Sorted First-Order Logic with Equality 7

3.2.2 Fixpoints 8

3.2.3 Partial Structures 8

3.2.4 Notation 9
3.3 Specification Language 10
3.3.1 Arithmetic and Aggregates 12
3.3.2 Inductive Definitions 13
3.4 Implementation 18
3.4.1 Computing Well-Founded Models of Inductive Definitions 19
3.4.2 Grounding 22
3.4.3 CNF Transformation 30
3.5 Experimental Evaluation 30
3.6 Related Work 31
3.7 Conclusion 34

4 Solving Modular Model Expansion **35**
4.1 Introduction 35
4.1.1 My Contributions 39
4.2 Background 40
4.2.1 Modular Systems 40
4.3 Computing Models of Modular Systems 40
4.3.1 Requirements on the Modules 41
4.3.2 Requirements on the Solver 46
4.3.3 Lazy Model Expansion Algorithm 47
4.4 Case Studies: Existing Frameworks 49
4.4.1 Modelling DPLL(T) 49
4.4.2 Modelling ILP Solvers 58
4.4.3 Modelling Constraint Answer Set Solvers 64
4.5 Related Work 72
4.6 Conclusion 73

5 Conclusion **75**

Appendix A Syntax of the Enfragmo System **76**
A.1 Problem Specification Grammar 76
A.2 Instance Description Grammar 80

Bibliography **82**

List of Tables

3.1 Step by step computation of $A^{-} = A^{\infty}_{P,\mathcal{B}}(\emptyset)$ 21
3.2 Answers to ϕ_1, ϕ_2, ϕ_3, t_1 and t_2 23
3.3 Complement of a False table. 24
3.4 A False table t from joining two False tables t_1 and t_2. 25
3.5 A False table t from joining a False table t_1 with a True table t_2. 25
3.6 A True table t from joining two True tables t_1 and t_2. 27
3.7 A True table t' from dividing a True table t using variables {x}. 29
3.8 Performance comparison of Enfragmo and other systems. 31

List of Figures

3.1 Enfragmo specification of K-colouring. 12

3.2 Enfragmo instance description for K-colouring. 12

3.3 Enfragmo specification for Knapsack variant. 14

3.4 Enfragmo specification for transitive closure of an edge relation. 16

3.5 Enfragmo specification of blocked N-queens. 17

3.6 Enfragmo specification and instance description for the recursive definitions of even numbers between 0 and 6. 19

4.1 Business Process Planner 36

4.2 Modular System $DPLL(T)_{\phi \wedge \psi}$ Representing the DPLL(T) System on Input Formula $\phi \wedge \psi$ 53

4.3 Facility Opening Problem Instance 60

4.4 Modular System Representing an ILP Solver 62

4.5 Planning with Cumulative Scheduling Problem Instance 67

Chapter 1

Introduction

Computationally hard search and optimization problems are ubiquitous in science, engineering and business. Examples include drug design, protein folding, phylogeny re-construction, hardware and software design, test case generation and verification, planning, timetabling, scheduling and so on. Often in practice, specialized domain expertise is required to design algorithms to solve these problems efficiently. Another way to solve search problems is to use the model-based problem solving approach. In this approach, the user only describes the properties of a solution, instead of designing an actual algorithm to construct the solutions. As no knowledge of programming is required, this approach considerably reduces specialized expertise required on the part of the user, making advanced solver technology accessible to a wider variety of users. The main goal of our research is to develop theoretical foundations for languages and systems for modelling and solving combinatorial search problems, and to build and demonstrate practical systems based on these foundations. We consider the *model expansion* (MX) task as the task representing the essence of search problems, where we are given an instance of a problem and search for a solution satisfying certain properties.

In this thesis, the Enfragmo system for representing and solving combinatorial search problems is presented. The system supports natural specification of problems by providing users with a rich language, based on an extension of first order logic. Enfragmo takes as input a problem specification and a problem instance and produces a solution to the problem if one exists. Our system is based on grounding, which is the task of producing a variable-free first-order formula representing the solution for the problem if one exists, with the aim of using a propositional satisfiability (SAT) solvers as the problem solving engine. The Enfragmo system uses classical logic to the greatest degree possible, while taking advantage of advances

in solver technology. However, because the specification language is high level, Enfragmo provides combinatorial problem solving capability to users who do not have expertise in use of SAT solvers or algorithms for solving combinatorial problems. Despite its rich syntax, experiments suggest that the performance of the Enfragmo system is comparable to that of state-of-the-art systems, e.g., IDP [15], Clingo [27], and DLV [15],.

Some of the search problems in practice, e.g., the task of constructing a logistics service provider relying on local service providers, are inherently modular. The MX-based framework is further extended to be able to represent a modular system. The framework treats specification languages equally and independently of their internal semantics and allows one to combine modules on an abstract model-theoretic level, independently from what languages are used for describing them. In this thesis, an algorithm for "solving" such modular MX systems is proposed, which takes a modular system of our definition, and generates its solutions. We show that our algorithm closely corresponds to what is done in practice in different areas such as Satisfiability Modulo Theories (SMT), Integer Linear Programming (ILP), and Answer Set Programming (ASP). However, we take combining modules to a new level by making our framework language-independent through a model-theoretic development.

The main contributions of the thesis are as follows. More specific contributions are stated in individual chapters.

1. We develop a model-based solver which provides users a rich language with clear semantics.
2. We design an abstract algorithm to solve the modular MX problem and formulated conditions on languages of individual modules to participate in the modular MX solving.
3. We show a similarity between our modular MX algorithm and the work of different solvers from different communities.

The work described in this thesis is based on joint work with Amir Aavani and Shahab Tasharrofi [58, 56, 5, 3, 4]. The contributions specific to the author are stated in the introduction Chapter 3 and Chapter 4.

The rest of the thesis is organized as follows. Chapter 2 presents some necessary background knowledge needed to read the thesis. The Enfragmo system is introduced in Chapter 3. The modular extension of the MX framework, the algorithm for solving a modular system, and the modular representations of existing systems are given in Chapter 4.

Chapter 2

Background

In this chapter, we formally introduce the concept of the *model expansion.* Before introducing the formal definition, we first review some necessary definitions. A vocabulary is a set τ of relation and function symbols, each with an associated arity. Constant symbols are zero-ary function symbols. A structure $\mathcal{A}$ for vocabulary τ (or, τ-structure) is a tuple containing a nonempty universe or domain A, and a relation (function) for each relation (function) symbol of τ. If R is a relation symbol of vocabulary τ, the relation corresponding to R in a τ-structure $\mathcal{A}$ is denoted $R^{\mathcal{A}}$. For example, we write

$$\mathcal{A} = (A; {R_1}^{\mathcal{A}}, ...{R_n}^{\mathcal{A}}, {c_1}^{\mathcal{A}}, ...{c_k}^{\mathcal{A}}, {f_1}^{\mathcal{A}}, ..., {f_m}^{\mathcal{A}}),$$

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

Let σ and τ be vocabularies, with $\sigma \subseteq \tau$, and let $\mathcal{A}$ be a σ-structure. A τ-structure $\mathcal{B}$ is an *expansion* [1] of $\mathcal{A}$ to τ if $A = B$ (i.e., their domains are the same), and for every relation symbol R and in σ, $R^{\mathcal{A}} = R^{\mathcal{B}}$ and for every function symbol f in σ, $f^{\mathcal{A}} = f^{\mathcal{B}}$.

2.1 Model Expansion

In [42], combinatorial search problems are formalized as the task of *model expansion (MX)*, the logical task of expanding a given (mathematical) structure with new relations. Formally,

[1] Expansion is a standard term in model theory. See, for example, [34].

the user axiomatizes the problem in some logic $\mathcal{L}$. We require the logic to have a standard model theory, i.e., its formulas must be interpretable over logical structures. This axiomatization relates an instance of the problem (a *finite structure*, i.e., a universe together with some relations and functions), and its solutions (certain *expansions* of that structure with new relations or functions). Logic $\mathcal{L}$ corresponds to a specification or modelling language, e.g., an extension of first-order logic such as FO(ID) [16], or an Answer Set Programming language, or a modelling language from the Constraint Programming community such as ESSENCE [26].

The task of model expansion for an arbitrary logic $\mathcal{L}$ (abbreviated $\mathcal{L}$-MX), is:

Model Expansion for logic $\mathcal{L}$

Given: *1.* An $\mathcal{L}$-formula ϕ with vocabulary $\sigma \cup \varepsilon$

2. A structure $\mathcal{A}$ for σ

Find: an expansion of $\mathcal{A}$, to $\sigma \cup \varepsilon$, that satisfies ϕ.

Thus, we expand the structure $\mathcal{A}$ with relations and functions to interpret ε, obtaining a model $\mathcal{B}$ of ϕ. We call σ, the vocabulary of $\mathcal{A}$, the *instance* vocabulary, and ε the *expansion* vocabulary.

Example 1 *The following formula ϕ of first order logic constitutes an MX specification for Graph 3-coloring:*

$$\begin{gathered}\forall x\ [(R(x) \vee B(x) \vee G(x)) \\ \wedge \neg((R(x) \wedge B(x)) \vee (R(x) \wedge G(x)) \vee (B(x) \wedge G(x)))] \\ \wedge\ \forall x \forall y\ [E(x,y) \supset (\neg(R(x) \wedge R(y)) \\ \wedge \neg(B(x) \wedge B(y)) \wedge \neg(G(x) \wedge G(y)))].\end{gathered}$$

A problem instance is a structure for vocabulary $\sigma = \{E\}$, i.e., a graph $\mathcal{A} = \mathcal{G} = (V; E)$. The task is to find an interpretation for the symbols of the expansion vocabulary $\varepsilon = \{R, B, G\}$ such that the expansion of $\mathcal{A}$ with these is a model of ϕ:

$$\underbrace{(\overbrace{V; E^{\mathcal{A}}}^{\mathcal{A}},\ R^{\mathcal{B}}, B^{\mathcal{B}}, G^{\mathcal{B}})}_{\mathcal{B}} \models \phi.$$

The structures $\mathcal{B}$ which satisfy ϕ are exactly the proper 3-colourings of $\mathcal{G}$.

Chapter 3

The Enfragmo System

3.1 Introduction

In practice, there are many computationally challenging search problems. In rare cases, practical application-specific software exists, but most often development of successful solution methods requires specialists to apply technology such as mathematical programming of constraint logic programming systems, or develop custom-refined implementations of general algorithms, such as branch and bound, simulated annealing, or reduction to SAT.

One goal of development of the Enfragmo system [1] is to provide a practical technology for solving combinatorial search problems, but one which would require considerably less specialized expertise on the part of the user, thus making technology for solving these problems accessible to a wider variety of users. In this approach, the user gives a precise specification of their search (or optimization) problem in a high-level declarative modelling language. A solver then takes this specification, together with an instance of the problem, and produces a solution to the problem (if there is one). We aim for using classical logic to the greatest degree possible, while taking advantage of advances in solver technology.

The model expansion (MX) [42], is in the foundation of our approach to solver construction. Users axiomatize their problems, formalized as model expansion, in some extension of classical logic. A problem instance, in this formalization, is a finite structure, and solutions to the instance are expansions of this structure that satisfy the specification formula. At present, our focus is on problems in the complexity class NP. For this case, the specification language is based on classical first-order logic (FO). Fagin's theorem [24] states that the problems which can be axiomatized in the existential fragment of second order logic ($\exists$SO)

are exactly those in NP, and thus the problems which can be axiomatized as FO MX are exactly the NP search problems.

Enfragmo's operation is based on grounding, which is the task of producing a variable-free first-order formula representing the expansions of the instance structure which satisfy the specification formula – in other words, the solutions for the instance. The ground formula is mapped to a propositional formula in conjunctive normal form (CNF), which is sent to a SAT solver. For any fixed FO formula, grounding can be carried out in polynomial time, so grounding provides a universal polytime reduction to SAT for problems in NP. The grounding algorithm for the Enfragmo system is based on the relational algebra based bottom-up grounding technique proposed in [43]. An important advantage in solving through grounding and transformation to SAT, or some other standard ground language, is that the performance of ground solvers is constantly being improved, and we can always select from the best solvers available.

Many interesting real-world problems cannot be conveniently expressed in pure FO MX, in particular if their natural descriptions involve arithmetic or properties defined inductively. Examples of the former include Knapsack and other problems involving weights or costs, while examples of the latter include the Traveling Salesman problem and other problems involving reachability. To address these issues, Enfragmo's specification language extends classical first order logic with arithmetic and aggregate operators, and includes a limited use of inductive definitions.

3.1.1 My Contributions

The work stated in this chapter is based on joint work with Amir Aavani [5, 3, 4]. The formal definition of grounding is contributed by Amir Aavani. The design and implementation of various grounding algorithms and CNF transformation algorithms are joint work between Amir Aavani and the author. The author individually designed and implemented the computation of well-founded model semantics of the inductive definitions for the Enfragmo system. The experiments shown in the chapter are also designed and run by the author.

3.2 Preliminaries

In this section, we formally define multi-sorted FO and present the concept of ordered structures. We assume the reader is familiar with the standard FO.

3.2.1 Ordered Structures and Multi-Sorted First-Order Logic with Equality

An ordered structure is one that contains a total order relation $\leq$ on its domain. While in many examples order is not needed, we find that having this relation as built-in saves a lot of effort in axiomatizing problems. In addition, it helps significantly in symmetry breaking, a technique to eliminate symmetrically equivalent solutions during search, thus indirectly reducing solving time. The order relation is also necessary for certain theoretical properties, such as defining a logic that captures the complexity class P. Since assuming ordered structures seems to cause no complications, we may always do so when convenient.

Next, we define multi-sorted FO. We assume that the vocabulary always contains designate predicate $\leq$. A vocabulary τ is a set of predicate and function symbols together with a set $\mathcal{S}$ of sorts (types). Each variable x, and constant symbol c is associated with a sort $S(x)$ and $S(c)$, respectively, where S is a function which maps a variable or a constant to one of the sorts in $\mathcal{S}$, and each function symbol f and predicate symbol p, with arity n, is associated with a tuple of sorts $T_f \in \mathcal{S}^{n+1}$ and $T_p \in \mathcal{S}^n$, respectively.

Let S be a function which maps a term to one of the sorts in $\mathcal{S}$. Then the set of terms over vocabulary τ is inductively defined by following rules:

- Every variable x is a term of sort $S(x)$.
- Every constant c is a term of sort $S(c)$.
- If $t_1, \cdots, t_n$ are terms of sort $S(t_1), \cdots, S(t_n)$, respectively, and f is a function symbol of arity n, associated with the tuple of sorts $(S(t_1), \cdots, S(t_n), s_f)$, then $f(t_1, \cdots, t_n)$ is a term of sort s_f.

The set of formulas over vocabulary τ is inductively defined by following rules:

- If $t_1, \cdots, t_n$ are terms of sort $S(t_1), \cdots, S(t_n)$, respectively, and p is a predicate symbol of arity n, associated with the tuple of sorts $(S(t_1), \cdots, S(t_n))$, then $p(t_1, \cdots, t_n)$ is a formula.

- If t_1 and t_2 are two terms of the same sort, then $t_1 = t_2$ and $t_1 \leq t_2$ are formulas.
- If ϕ and ψ are formulas, x is a variable, then $\neg\phi$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $\forall x : S_x\ \phi$, and $\exists x : S_x\ \phi$ are all formulas.

We use the abbreviations $(\phi \to \psi)$ for $(\neg\phi \vee \psi)$, and $(\phi \leftrightarrow \psi)$ for $((\phi \to \psi) \wedge (\psi \to \phi))$. We sometimes also use $\forall x : S_x \leq y\ \phi$ as the shorthand for $\forall x : S_x\ (x \leq y \to \phi)$, and similarly for $\exists$. We may drop the parentheses as long as the *unique readability* of formulas is still assured.

The formulas obtained by the first two rules are called atomic formulas. A literal is either an atomic formula or its negation. An occurrence of a subformula in a formula is positive (resp. negative) if it occurs in the scope of an even (resp. odd) number of negations.

An interpretation for a vocabulary τ is provided by a *structure* $\mathcal{A}$, which consists of a domain A_s for each sort s in $\mathcal{S}$, a domain element $c^{\mathcal{A}}$ for each constant c, a relation $p^{\mathcal{A}}$ in $S(t_1)^{\mathcal{A}} \times \cdots \times S(t_n)^{\mathcal{A}}$ for each predicate symbol p of arity n, associated with the tuple of sorts $(S(t_1), \cdots, S(t_n))$, and a function $f^{\mathcal{A}}$ of type $S(t_1)^{\mathcal{A}} \times \cdots \times S(t_n)^{\mathcal{A}} \to s_f{}^{\mathcal{A}}$ for each function symbol f of arity n, associated with the tuple of sorts $(S(t_1), \cdots, S(t_n), s_f)$. The satisfaction relation $\models$ is defined as usual except that, informally speaking, the quantifications are over the domains of corresponding variable sorts.

3.2.2 Fixpoints

Let C be a set and $Pow(C)$ be the power set of C. Then, a function $F : Pow(C) \to Pow(C)$ gives rise to a sequence of sets:

$$\emptyset, F(\emptyset), F(F(\emptyset)), \cdots,$$

where $F^0 = \emptyset$ and $F^{i+1} = F(F^i)$. We call F^i the i-th *stage* of F. If there is an $i_0 \in \mathbb{N}$ such that $F^{i_0+1} = F^{i_0}$, then $F^i = F^{i_0}$ for all $i \geq i_0$. We call F^{i_0} the fixpoint of F, denoted by F^∞. The function F is *monotone* (resp. *antimonotone*) if for all $X, Y \in C$, $X \subseteq Y$ implies $F(X) \subseteq F(Y)$ (resp. $F(Y) \subseteq F(X)$). If F is monotone, $F^i(\emptyset) \subseteq F^{i+1}(\emptyset)$ for all $i \in \mathbb{N}$, and we call $F^\infty(\emptyset)$ the least (w.r.t. the set inclusion) fixpoint of F.

3.2.3 Partial Structures

To demonstrate the model computation procedure of the inductive definition program in Enfragmo (see Section 3.4.1), we need the concept of partial structures. A partial structure

is a structure that may contain unknown values. Here we talk about regular (not multi-sorted) structures. The generalization to multi-sorted structures is straightforward.

Definition 1 (Partial Structure) *We say $\mathcal{B}$ is a τ_p-*partial structure *over vocabulary τ if:*

1. *$\tau_p \subseteq \tau$,*
2. *$\mathcal{B}$ gives a total interpretation to symbols in $\tau \backslash \tau_p$ and,*
3. *for each n-ary symbol R in τ_p, $\mathcal{B}$ interprets R using two sets R^+ and R^- such that $R^+ \cap R^- = \emptyset$, and $R^+ \cup R^- \neq (dom(\mathcal{B}))^n$.*

We say that τ_p is the partial vocabulary of $\mathcal{B}$. If $\tau_p = \emptyset$, then $\mathcal{B}$ is total. For two partial structures $\mathcal{B}$ and $\mathcal{B}'$ over the same vocabulary and domain, $\mathcal{B}'$ extends *$\mathcal{B}$ if all unknowns in $\mathcal{B}'$ are also unknowns in $\mathcal{B}$, i.e., $\mathcal{B}'$ has at least as much information as $\mathcal{B}$.*

Example 2 *Consider a structure $\mathcal{B}$ with domain $\{0, 1, 2\}$ for vocabulary $\{I, R\}$, where I and R are unary relations, and $I^{\mathcal{B}} = \{\langle 0\rangle, \langle 1\rangle\}$, $\langle 0\rangle \in R^{\mathcal{B}}$, and $\langle 1\rangle \notin R^{\mathcal{B}}$, but it is unknown whether $\langle 2\rangle \in R^{\mathcal{B}}$ or $\langle 2\rangle \notin R^{\mathcal{B}}$. Then $\mathcal{B}$ is a $\{R\}$-partial structure over vocabulary $\{I, R\}$ where ${R^+}^{\mathcal{B}} = \{\langle 0\rangle\}$ and ${R^-}^{\mathcal{B}} = \{\langle 1\rangle\}$.*

For an atomic formula ϕ, we say $\mathcal{B} \models \phi$ (resp. $\mathcal{B} \models \neg\phi$) if ϕ is true (resp. false) in the interpretation of the total part of vocabulary $(\tau \setminus \tau_p)$ or in R^+. The satisfiability relation for compound formulas could be defined recursively in the standard way. Note that for partial structures, $\mathcal{B} \models \neg\phi$ and $\mathcal{B} \not\models \phi$ may be different. We call a ε-partial structure $\mathcal{B}$ over $\sigma \cup \varepsilon$ the *empty expansion* of σ-structure $\mathcal{A}$, if $\mathcal{B}$ agrees with $\mathcal{A}$ over σ but $R^+ = R^- = \emptyset$ for all $R \in \varepsilon$. When we talk about a τ_p-partial structure, in the MX context, τ_p is always a subset of ε.

3.2.4 Notation

Let $\mathcal{B}$ be a σ-structure, $\tau \subseteq \sigma$ and S be a set of literals with the set of non-logical symbols τ, then

1. A complement of a literal l is the literal corresponding to the negation of l. More precisely, if l is an atomic formula a, then the complement of l is $\neg a$, and if l is the negation of an atomic formula, i.e., $\neg a$, then the complement of l is a.

2. $\neg S$ denotes the set $\{\neg l \mid l \in S\}$.

3. $\tau^{\mathcal{B}}$ denotes $\bigcup_{p \in \tau} p^{\mathcal{B}}$.

4. $U_{\mathcal{B}}(S)$ denotes the set containing all possible instantiations of predicate symbols in τ in terms of $\mathcal{B}$, i.e., $U_{\mathcal{B}}(S) = \bigcup_{predicate\ p \in \tau} dom(\mathcal{B})^{arity(p)}$.

5. If S is a set of atoms, the *conjugate* over $\mathcal{B}$ is the negative set $\overline{S} = \neg U_{\mathcal{B}}(S) \setminus \neg S$.

6. If S is a set of negative literals, the *conjugate* over $\mathcal{B}$ is the positive set $\overline{S} = U_{\mathcal{B}}(S) \setminus \neg S$.

7. We sometimes use a (partial) structure $\mathcal{B}$, which gives an interpretation to vocabulary τ, as the set of literals which are true according to $\mathcal{B}$. For example, for a τ_p partial structure $\mathcal{B}$ such that $\tau = \{P, R\}$, $\tau_p = \{P\}$, $dom(\mathcal{B}) = \{1, 2\}$, $R^{\mathcal{B}} = \{1\}$, $P^{+\mathcal{B}} = \emptyset$ and $P^{-\mathcal{B}} = \{1\}$, we may represent $\mathcal{B}$ using following set of literals:

$$S = \{R(1), \neg R(2), \neg P(1)\}.$$

The operator $structure_{\mathcal{B}}(.)$ may be used to recover the structure $\mathcal{B}$ from S.

3.3 Specification Language

In this thesis, our focus is on problems in the complexity class NP. For this case, the modelling language is based on classical first-order logic (FO). Fagin's theorem [24] states that the problems which can be axiomatized in the existential fragment of second order logic (∃SO) [1] are exactly those in NP, i.e., FO-MX captures NP. This result makes FO a good specification language for modelling and solving NP search problems.

In [42], the authors argue that an important property for a specification language is capturing a complexity class. As FO-MX captures NP, we know that:

1. FO-MX can express every problem in NP - which gives the user an assurance of universality of the language for the complexity class NP,

[1]Informally, ∃SO is an extension of FO which have the variables in FO ranging over the individuals, as well as additional variables that range over a set of individuals with the restriction that these variables can only be quantified over the existential quantifier.

2. no more than NP can be expressed - thus solving can be achieved by a universal polytime reduction, called *grounding*, to some well-studied NP-complete problems like SAT and CSP.

The specification language of the Enfragmo system is based on multi-sorted classical first-order logic (see section 3.2.1), extended with inductive definitions, arithmetic functions, and aggregate operators. We will illustrate the language with examples, and give brief discussions of some major features. For the full description of the input language, please refer to the Enfragmo manual, which is available from [1]. The exact syntax of specification and instance description is also available in the Appendix.

Example 3 shows a sample specification and instance description for the Enfragmo system. As shown in Figure 3.1, an Enfragmo specification consists of four main sections, delineated by keywords. The `GIVEN:` section defines the types and vocabulary used in the specification. The `FIND:` section identifies the vocabulary symbols for which the solver must find interpretations, that is, the functions and relations which will constitute a solution. Interpretations of the remaining vocabulary symbols are given by the problem instance. The third part consists of one or more `PHASE:` sections, each of which contains an optional `FIXPOINT:` part, which provides an inductive definition, followed by a `SATISFYING:` part, which consists of a set of sentences in the extended first order logic. If there are multiple `PHASE:` sections, they define a sequence of expansions. One way such a sequence can be used is to carry out a kind of pre-processing or post-processing, which may support more convenient axiomatizations or more efficient solving. An example is provided in the section on inductive definitions below. Finally, the `PRINT:` section identifies relations that are to be displayed, if a solution is found.

Figure 3.2 shows a sample instance description for the graph k-colouring problem. An instance description contains the domains of types and the interpretations of instance relations and (total) functions. Domains may be given as a range of integers (e.g., `Vtx [1..5]`) or as an enumerated list of constant symbols (e.g., `Clr ['red', 'green', 'blue']`). For a range of integers the order is numerical; for an enumerated list it is as given. Thus, it imposes a total order in the set of domain elements. Interpretations of relations are given as a set of space-separated tuples, with elements of tuples being comma-separated.

Example 3 (Graph K-Colouring) *To axiomatize Graph K-Colouring, we introduce two sorts, vertices and colours. The axiomatization says that there is a binary relation Colour*

which must be a proper colouring of the vertices. The corresponding Enfragmo specification is given in Figure 3.1, and an instance description is given in Figure 3.2. [2]

GIVEN:
  TYPES: Vtx Clr;
  PREDICATES: Edge(Vtx,Vtx), Colour(Vtx,Clr);
FIND: Colour;
  // expansion predicate(s) are listed under FIND
  //(instance predicates are those that are not expansion)
PHASE:
  SATISFYING:
    // every vertex has at least one colour
    $\forall\ v : Vtx\ \exists c : Clr\ Colour(v, c);$
    // no vertex has more than one colour
    $\forall v : Vtx\ \forall c : Clr\ (Colour(v, c)\ \rightarrow\ \neg\exists c2 : Clr\ (c2\ <\ c\ \wedge\ Colour(v, c2)));$
    // no two vertices of the same colour are adjacent
    $\forall u : Vtx\ \forall v : Vtx\ \forall c : Clr\ ((Colour(u, c)\ \wedge\ Colour(v, c))\ \rightarrow\ \neg Edge(u, v));$
PRINT: Colour; // solution can be printed

Figure 3.1: Enfragmo specification of K-colouring.

TYPE Vtx [1..5]
TYPE Clr ['red', 'green', 'blue']

PREDICATE Edge
  (1, 2) (1, 4) (1, 5) (2, 3) (2, 5) (3, 4) (3, 5)

Figure 3.2: Enfragmo instance description for K-colouring.

Next, we discuss some major features of the Enfragmo specification language.

3.3.1 Arithmetic and Aggregates

The theoretical foundations of MX with arithmetic have been studied in [59, 54, 55]. Here, we focus on a fragment that is implemented. Enfragmo specifications have two kinds of types: integer types and enumerated types. Terms of integer types may use the arithmetic

[2] *In Figure 3.1, the symbol $<$ is used instead of $\neq$ in the second formula to reduce the search space.*

functions $+$, $-$, $*$, and $ABS(\cdot)$, which have their standard meaning. Arithmetic terms also include the aggregate operators maximum, minimum, sum, and cardinality. In the following, if $\phi(\bar{x})$ is a formula with free variables $\bar{x}$, then $\phi^{\mathcal{B}}[\bar{a}]$ denotes the truth value of ϕ in structure $\mathcal{B}$ when the variables $\bar{x}$ denote the domain elements $\bar{a}$, and similarly for terms $t(\bar{x})$. The aggregate terms are defined, as follows, with respect to a structure $\mathcal{B}$ in which the formula containing the term is true.

- $Max_{\bar{x}}\{t(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y}); d_M\}$ denotes, for any instantiation $\bar{b}$ for $\bar{y}$, the maximum value obtained by $t^{\mathcal{B}}[\bar{a},\bar{b}]$ over instantiations $\bar{a}$ for $\bar{x}$ for which $\phi^{\mathcal{B}}[\bar{a},\bar{b}]$ is true, or d_M (the default value) if there are none.
- $Min_{\bar{x}}\{t(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y}); d_m\}$ is defined dually to Max.
- $Sum_{\bar{x}}\{t(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y})\}$, denotes, for any instantiation $\bar{b}$ for $\bar{y}$, the sum of all values $t^{\mathcal{B}}[\bar{a},\bar{b}]$ over instantiations $\bar{a}$ for $\bar{x}$ for which $\phi^{\mathcal{B}}[\bar{a},\bar{b}]$ is true.
- $Count_{\bar{x}}\{\phi(\bar{x},\bar{y})\}$ denotes, for any instantiation $\bar{b}$ for $\bar{y}$, the number of tuples $\bar{a}$ for which $\phi^{\mathcal{B}}[\bar{a},\bar{b}]$ is true.

Example 4 illustrates use of arithmetic terms, including sum and count aggregates.

Example 4 (A Knapsack Problem Variant) *Consider the following variation of the knapsack problem: We are given a set of items (loads) $L = \{l_1, \cdots, l_n\}$, each with an integer weight $W(l_i)$, and m knapsacks $K = \{k_1, \cdots, k_m\}$. The task is to put the n items into the m knapsacks, while satisfying the following constraints. 1) Certain items must go into preassigned knapsacks, as specified by the binary instance predicate P; 2) H of the m knapsacks are high capacity, and can hold items with total weight Cap_H, while the remainder have capacity Cap_L; 3) No knapsack may contain two items with weights that differ by more than D. Each of Cap_H, Cap_L and D is an instance function with arity zero, i.e. a given constant. An Enfragmo specification for this problem is given in Figure 3.3. Q is the mapping of items to knapsacks that must be constructed.*

3.3.2 Inductive Definitions

Inductive properties are not easily expressed as FO-MX, and the methods for expressing them tend to produce specifications which, in our experience, are not well handled by current

```
GIVEN:
  TYPES: Item Knaps;
  INTTYPES: Weight ItemCount;
  PREDICATES: P(Item, Knaps) Q(Item, Knaps);
  FUNCTIONS:
    W(Item): Weight
    Cap_H(): Weight
    Cap_L(): Weight
    D(): Weight
    H(): ItemCount;
FIND: Q;
PHASE:
  SATISFYING:
    // Q is a function mapping items to knapsacks
    ∀l : Item ∃k : Knaps Q(l, k);
    ∀l : Item ∀k1 : Knaps ∀k2 : Knaps ((Q(l, k1) ∧ Q(l, k2)) → k1 = k2);
    // Q agrees with the pre-assignment P
    ∀l : Item ∀k : Knaps (P(l, k) → Q(l, k));
    // The total weight in each knapsack is at most Cap_H
    ∀k : Knaps SUM{l : Item; W(l); Q(l, k)} ≤ Cap_H();
    // At most H knapsacks have total weight greater than Cap_L
    COUNT{k : Knaps; SUM{l : Item; W(l); Q(l, k)} > Cap_L()} ≤ H();
    // Items in a knapsack differ in weight by at most D
    ∀k : Knaps ∀l1 : Item ∀l2 : Item ((Q(l1, k) ∧ Q(l2, k)) → ABS(W(l1) − W(l2)) ≤ D());
PRINT: Q;
```

Figure 3.3: Enfragmo specification for Knapsack variant.

solvers. Therefore, for practical purposes, it is necessary to extend FO with inductive definitions.

In our view, it is more appropriate to extend classical logic with a mechanism for (non-monotone) induction than to base an entire language or system on a non-monotonic logic. The reason is that the classical semantics is much simpler to work with, both for language users and system designers. In particular, classical logic is modular, that is, combining FO specifications is easy, because no special conditions are required to ensure the semantic content of a formula remains unchanged in combination with others. Since the large majority of constraints in most applications are classical - that is, do not involve minimization or closure - we may use the more complex semantics of induction when and to the extent we

need it. In contrast, with a purely non-monotonic logic, sometimes we must simulate the classical semantics using the more complex non-classical semantics.

The logic FO(ID) is a logic which augments FO with a non-classical construction to represent induction, developed in [17]. The construct allows for natural modelling of monotone and non-monotone inductive definitions, including iterated induction and induction over a well-founded order. It is proposed as the logic of choice for modelling NP-search problems as model expansion in [42]. It is also shown in [42] that FO(ID)-MX still captures NP. A very efficient system capable of dealing with general inductive definitions has been developed at KU Leuven [68]. Their solver, The IDP System, is able to support the full language of FO(ID).

The formulas of FO(ID) are constructed from atomic formulas $P(\bar{t})$, and definitions Δ, closed under the standard use of $\wedge, \vee, \neg, \forall, \exists$. A definition Δ is a set of rules as illustrated in example 5. The syntax is quite general: the body of the rules may be unrestricted FO formulas. Disjunctions of definitions, multiple definitions for the same predicate, etc., are all allowed. Predicates appearing in the heads of the rules are called defined, and those which are not defined are called open.

The "$\leftarrow$" operator in rules of inductive definitions is the *definitional implication*, with semantics provided by the well-founded semantics from logic programming. The well-founded model of an FO formula partitions the model into three parts: true, false and unknown. If an atom is true (resp. false) in the well-founded model of an FO formula ϕ, then it is true (resp. false) in every model of ϕ. More details can be found in [66]. A structure $\mathcal{B}$ satisfies a set of rules Δ iff $\mathcal{B}$ is the 2-valued well-founded model of Δ.

Enfragmo implements a fragment of the logic FO(ID), where definitions can only appear in conjunctions with the rest of the axiomatization. Inductive definitions are specified under the `FIXPOINT` keyword followed by a comma separated list of predicates to be defined in the paranthesis. Each definition is wrapped using curly braces under the `DEFINE` keyword. A definition is composed of one or more rules with variable quantifications. The head of each rule is a predicate to be defined, and the body is a formula of FO extended with functions, arithmetic operations and aggregates. Recall that the Enfragmo system grounds the input program into a standard SAT instance, but grounding inductive definitions into a SAT instance could be very hard and may produce very ineffective instances. In contrast to the full version of inductive definitions implemented in [69, 67, 38], no expansion predicates in the body are allowed in our system, i.e., the open predicates in a definition must be

instance predicates, or have been constructed explicitly in a previous `Phase:` section. Even this limited form has proved to be very useful in practice. These definitions can be used to efficiently compute useful information, such as a bound or partial solution, which can be used later to help solve a problem more efficiently. Examples of such usage are shown in example 6 and 7.

Example 5 (Transitive Closure of an Edge Relation) *Figure 3.4 shows the Enfragmo specification for computing the transitive closure of edge relation in the given graph.*

```
GIVEN:
   TYPES: SVtx ;
   PREDICATES: Edge (SVtx, SVtx) TC (SVtx, SVtx) ;
FIND: ;
PHASE:
   FIXPOINT (TC) :
      DEFINE
      {
        x : SVtx  y : SVtx TC(x, y) ← Edge(x, y)
        x : SVtx  y : SVtx TC(x, y) ← ∃z : SVtx (TC(x, z) ∧ Edge(z, y))
      }
   ;
PRINT : TC ;
```

Figure 3.4: Enfragmo specification for transitive closure of an edge relation.

Example 6 (Blocked N-Queens) *The blocked N-queens problem is a variant of the N-queens problem. In this problem, we are given a $N \times N$ board and N queens. Some of the cells on the board are blocked. The task is to place N queens on some non-blocked cells such that no two queens are able to attack each other, i.e., no two queens are placed in the same row, column, or diagonal. Figure 3.5 shows an Enfragmo specification for blocked N-queens problem. In the specification, an inductive definition is used to first compute pairs of non-blocked cells that are in the same diagonal.*

Example 7 (Graph K-Colouring:Continued) *Graph Colouring can be solved more efficiently by having inductive definitions in an initial group of phases compute a maximal clique in the graph, and pre-assign distinct colours to the vertices in that clique. (A further improvement might be to construct a maximal clique containing a vertex of maximum*

```
GIVEN:
   INTTYPES: Num;
   PREDICATES:
      Block (Num, Num)
      Queen (Num, Num)
      Diag (Num, Num, Num, Num)
   ;
FIND: Queen ;
PHASE:
   FIXPOINT (Diag) :
      DEFINE
      {∀x1 : Num ∀y1 : Num ∀x2 : Num ∀y2 : Num Diag(x1, y1, x2, y2) ←
         x1 < x2 ∧ ¬Block(x1, y1) ∧ ¬Block(x2, y2) ∧ ABS(x1 − x2) = ABS(y1 − y2)}
   ;
   SATISFYING:
      //queens can only be placed in non-blocked cells.
      ∀r : Num ∀c : Num (Queen(r, c) → ¬Block(r, c));
      //each column should contain exactly one queen.
      ∀r : Num COUNT{c : Num; Queen(r, c) ∧ ¬Block(r, c)} = 1;
      //each row should contain exactly one queen.
      ∀c : Num COUNT{r : Num; Queen(r, c) ∧ ¬Block(r, c)} = 1;
      //two non-blocked cells in the same diagonal cannot both contain queens.
      ∀x1 : Num ∀y1 : Num ∀x2 : Num ∀y2 : Num
                              (Diag(x1, y1, x2, y2) → ¬(Queen(x1, y1) ∧ Queen(x2, y2)));
PRINT: Queen ;
```

Figure 3.5: Enfragmo specification of blocked N-queens.

degree.) Then a final phase can construct a colouring of the graph restricted by the pre-computed colouring of the large clique. We use this technique in the experiments shown in section 3.5.

In addition to the semantics from the language of FO(ID), the inflationary fixpoint semantics [37] is also implemented in Enfragmo. Given an inductive definition, we construct the inflationary fixpoint by following iterative procedure: starting from the empty interpretation, we iteratively derive new information using the facts derived from the previous iterations. The exact definition can be found from [37]. To use this implementation, one only needs to replace the `DEFINE` keyword in the inductive definition with `INFLATE`. Note

that for the definitions involving negation in the body, we cannot expect the inflationary fixpoint operator to give a desired result. For example, it does not give a desired output on the recursive definition of even numbers shown in the Example 8, i.e., the interpretation of the predicate $Even$ will contain all the numbers between 0 and n in the first iteration, because $Even$ is initially empty. On the other hand, under the well-founded model semantics described above, we can see that the interpretation of $Even$ containing the set of even numbers between 0 and n is indeed the 2-valued well-founded model of the program. We describe the algorithm used to compute this well-founded model in Section 3.4.1.

Note that when none of the atomic formulas $P(\bar{t})$ appear negatively in the definitions Δ, the inflationary fixpoint semantics and the well-founded model semantics coincide. In this case, Enfragmo uses the inflationary fixpoint implementation because it is more efficient.

Example 8 (Recursive definition of the even numbers)
Figure 3.6 shows the Enfragmo specification and instance description for the recursive definitions of even numbers below:

- *0 is even.*
- *(n+1) is even if and only if n is not even.*

3.4 Implementation

Enfragmo's input is a problem specification together with a description of an instance, stated in the language described in Section 3.3. (Eventually, Enfragmo may be extended so that instances may be retrieved by queries to a database or other source.) The phases in the specification are solved one-by-one, in the order written. For each phase, any predicates defined by an inductive definition are computed, and then the satisfying phase is solved by grounding. This in turn involves three stages: 1) grounding each formula with respect to the instance and information computed from previous phases to produce a ground FO formula representing the solutions; 2) the ground formula is transformed to a propositional CNF formula; 3) a SAT solver is called on the CNF formula; 4) if the SAT solver reports a satisfying assignment, it is mapped back to a description of a solution in the vocabulary of the specification.

```
GIVEN:
  INTTYPES: int;
  PREDICATES: Even(int);
FIND: ;
PHASE:
  FIXPOINT(Even):
    DEFINE
    {
      x : int  :  Even(x)  ←  x = MIN[int]
      x : int  :  Even(x)  ←  ∃y : int  :  (¬Even(y)  ∧  x = y + 1)
    }
  ;
PRINT: Even;

TYPE int[0..6]
```

Figure 3.6: Enfragmo specification and instance description for the recursive definitions of even numbers between 0 and 6.

3.4.1 Computing Well-Founded Models of Inductive Definitions

Recall that a structure $\mathcal{B}$ is a model of the inductive definition program P if and only if $\mathcal{B}$ is the 2-valued well-founded model of the program P, or equivalently, the unique well-founded model is a total model of the program P. The well-founded partial model is defined non-constructively in [66]. We implement the concept using the idea of *alternating fixpoint* of logic programs [65], but in the MX setting. The idea is to define an antimonotone operator on a set of negative facts, which go back and forth between underestimates and overestimates of the negative portion of the well-founded model and eventually converges to the exact set of negative facts in the well-founded model. In this section, we give the formal definition of *alternating fixpoint partial model* and its construction in the MX setting.

The *immediate consequence operator* is an operator whose fixpoint has been used as a semantics of logic programs [64]. In [65], the author extends the definition of *immediate consequence operator* to rules with negative literals. Here, we further extend it to allow an arbitrary FO formula as the body of a rule. Recall from the Section 3.2.4 that we sometimes use a (partial) structure $\mathcal{B}$ as the set of literals which are true according to $\mathcal{B}$. And we use the operator $structure_{\mathcal{B}}(S)$ to recover the structure $\mathcal{B}$ from S.

Definition 2 (Immediate Consequence Operator) *The* immediate consequence operator *for an inductive definition P over a (partial) structure $\mathcal{B}$ is the function $C_{P,\mathcal{B}}(S^+, S^-)$, which takes two sets with each set containing positive and negative literals, respectively, and outputs a set of positive literals. A literal l is in the output if and only if P contains a (ground) rule whose head is l and $structure_{\mathcal{B}}(S^+ \cup S^-)$ satisfies the body of the rule.*

Now, we define the alternating fixpoint operator used to compute the well-founded partial model of the inductive definition.

Definition 3 (Alternating Fixpoint Operator in MX Setting) *Let σ-structure $\mathcal{A}$ be the instance structure, $\mathcal{B}$ be a (partial) structure, and P be an inductive definition. Let $I^+ = \sigma^{\mathcal{A}}$, and $I^- = \overline{I^+} = \neg U_{\mathcal{A}}(I^+) \setminus \neg I^+$ [3]. Define an operator $S_{P,\mathcal{B}}$ on a set of negative literals such that*

$$S_{P,\mathcal{B}}(S^-) = T^{\infty}_{P,\mathcal{B}}(S^-),$$

where $T_{P,\mathcal{B}}(S^-) = C_{P,\mathcal{B}}(I^+, S^- \cup I^-)$. Note that the operator $T_{P,\mathcal{B}}$ is monotone, thus the fixpoint always exists. Based on the definition of the operator $C_{P,\mathcal{B}}$, the output of $S_{P,\mathcal{B}}$ is a set of positive literals. We convert it into a set of negative literals by taking the conjugate, i.e., define another operator $\widetilde{S}_{P,\mathcal{B}}(S^-) = \overline{S_{P,\mathcal{B}}(S^-)} = U_{\mathcal{A}}(S_{P,\mathcal{B}}(S^-)) \setminus \neg S_{P,\mathcal{B}}(S^-)$.

Finally, we define the alternating fixpoint operator *$A_{P,\mathcal{B}}$ on a set of negative literals such that*

$$A_{P,\mathcal{B}}(S^-) = \widetilde{S}_{P,\mathcal{B}}(\widetilde{S}_{P,\mathcal{B}}(S^-)).$$

Note that the operator $\widetilde{S}_{P,\mathcal{B}}$ is antimonotone. So by applying it twice, we get a monotone operator $A_{P,\mathcal{B}}$. We call its least fixpoint $A^- = A^{\infty}_{P,\mathcal{B}}(\emptyset)$ the alternating fixpoint *of the program P over the (partial) structure $\mathcal{B}$.*

Definition 4 (Alternating Fixpoint Partial Model) *Let A^- be the least fixpoint as defined above and $A^+ = S_{P,\mathcal{B}}(A^-)$. We call a partial structure $\mathcal{B}$ the* alternating fixpoint partial model *of program P if and only if $\mathcal{B} = structure_{\mathcal{B}}(A^+ \cup A^-)$.*

The alternating fixpoint partial model is equivalent to the well founded partial model [65]. In Enfragmo, the alternating fixpoint partial model is constructed using the above fixpoint computation procedure, and the inductive definition is considered satisfiable if and

[3] *The literals in both I^+ and I^- are always true according to the instance structure.*

i	S_i^-	$S_{P,\mathcal{B}}(S_i^-)$	$\widetilde{S}_{P,\mathcal{B}}(S_i^-)$	$S_{P,\mathcal{B}}(\widetilde{S}_{P,\mathcal{B}}(S_i^-))$
0	$\emptyset$	0	$\neg 1, \neg 2, \neg 3, \neg 4, \neg 5, \neg 6$	$0, 2, 3, 4, 5, 6$
1	$\neg 1$	$0, 2$	$\neg 1, \neg 3, \neg 4, \neg 5, \neg 6$	$0, 2, 4, 5, 6$
2	$\neg 1, \neg 3$	$0, 2, 4$	$\neg 1, \neg 3, \neg 5, \neg 6$	$0, 2, 4, 6$
3	$\neg 1, \neg 3, \neg 5$	$0, 2, 4, 6$	$\neg 1, \neg 3, \neg 5$	$0, 2, 4, 6$
4	$\neg\mathbf{1}, \neg\mathbf{3}, \neg\mathbf{5}$	$0, 2, 4, 6$	$\neg 1, \neg 3, \neg 5$	$0, 2, 4, 6$

Table 3.1: Step by step computation of $A^- = A^{\infty}_{P,\mathcal{B}}(\emptyset)$

only if the computed model is a total model. Note that all the operations above are straightforward to implement except for the *immediate consequence operator*. In logic programming, this is trivial since the body of each rule only consists of conjunction of literals. But in Enfragmo, any FO formula can appear as the body of a rule. In Enfragmo, it is implemented by the *grounding* technique described in section 3.4.2, i.e., the output of the immediate consequence operator is the union of the grounding results of the body in each rule, using the structure $structure_{\mathcal{B}}(S^+ \cup S^-)$ (See definition 3.2.4).

The following example illustrates the procedure to compute the alternating fixpoint partial model of inductive definition program shown in Example 8.

Example 9 (Recursive Definition of Even Numbers Continued) *This example illustrates how the alternating fixpoint partial model of the inductive definition program in Example 8 is actually constructed. The step by step computation of the alternating fixpoint is shown in Table 3.1. In Table 3.1, we omit the predicate symbol Even, e.g., instead of* $\neg Even(1)$, *we write* $\neg 1$. *Since the alternating fixpoint partial model is a total model, the inductive definition in example 8 has a model $\mathcal{B}$ which is an expansion of the instance structure $\mathcal{A}$ by the relation Even such that:*

$$Even^{\mathcal{B}} = \{0, 2, 4, 6\}.$$

A preliminary version of the inductive definition is implemented in Enfragmo using the alternating fixpoint computation procedure just described. There are some improvements that could be achieved. In the current implementation, we do grounding each time we apply the immediate consequence operator. Some smart data structure is needed to apply the grounding only once in the beginning, and update the data structure based on different assignments.

3.4.2 Grounding

Enfragmo computes a grounding of a formula bottom up, in a process analogous to bottom-up evaluation of a database query using the relational algebra. It is important to notice that the model expansion problem is very different from a query evaluation problem. In model expansion context, there are formulas and sub-formulas involving expansion predicates/functions, which cannot be evaluated, while in a query processing context, every formula can be evaluated as either true or false.

In Enfragmo, an extension of the relational algebra is used, in which a formula is associated with each tuple. This technique is first proposed in [43]. A tuple contains domain elements, and the associated formula is (equivalent to) a ground instance of a sub-formula of the specification formula, with variables instantiated by constants denoting the domain elements in the tuple. An "answer" to a sub-formula of the specification formula is an extended table representing all instantiations of the sub-formula. The answer for a sentence consists of an empty tuple associated with a formula since a sentence does not have any free variables. The formula is a grounding of the sentence with respect to the instance. Similarly, an answer to a term is a set of triples, each consisting of an instantiation of the arguments, a value the term may denote, and a formula. To compute answers for compound formulas and terms, we use the relational algebra operations corresponding to each connective and quantifier in FO, as follows: complement (negation); join (conjunction); union (disjunction), projection (existential quantification); division or quotient (universal quantification). For example, to compute the answer to the compound formula $\phi \wedge \psi$, we first compute the answers to its subformulas ϕ and ψ, and join the two extended tables (answers) together. The join of answers to formulas is the same as the join of tables in relational algebra, except that we also join (conjunct) the formulas associated to the tuples. For another example, to compute the answer to the compound term $t_1 + t_2$, we first compute the answers to the subterms t_1 and t_2, and then join the two answer tables. The join of answers to terms is the same as the join of answers to formulas, except that the corresponding output value of a compound term is determined based on the semantic of the operator used to form the compound term (plus in the example), and the output values of its subterms. The exact construction of the answers to the formulas and terms can be found in [5] and [2]. Example 10 shows the answers to some sample formulas and terms.

Example 10 *Let $\sigma = \{P, f\}$ and $\varepsilon = \{E\}$, and let $\mathcal{A}$ be a σ-structure with $dom(\mathcal{A}) = \{1, 2\}$*

and $P^{\mathcal{A}} = \{(1,1,1),(2,2,2)\}$*, and* $f^{\mathcal{A}} = \{(1:2),(2:1)\}$*. Answers to* $\phi_1 \equiv (P(x,y,z) \wedge E(x,y) \wedge E(y,z))$*,* $\phi_2 \equiv \exists z \phi_1$*,* $\phi_3 \equiv \exists x \exists y \phi_2$*,* $t_1 \equiv f(x)$*, and* $t_2 \equiv t_1 + y$ *are demonstrated in Table 3.2. In Table 3.2, the tuples associated with the formula False are omitted.*

Note that some of the output values in the answer to the arithmetic term t_2 in example 3.2 is outside of the domain. This can be problematic since the arithmetic operations may potentially involve infinite domains, and if we don't deal with the problem carefully, the expressive power of the language will be easily become out of control. The problem is addressed by a technique described in [59], in which the MX framework is embedded with an infinite background structure. The authors show that their notion of *embedded MX* still captures NP. We use this technique in the Enfragmo system to deal with arithmetical constructs with possibly infinite domains.

Efficiency of this grounding method requires using suitable data structures. For efficiency, all formulas in computed answers are represented in a directed acyclic graph (DAG) which is constructed as the operations of the algebra are applied. We memorize the previously generated formulas and their structures to prevent generating the same subformulas more than once. Next we briefly describe some aspects of the implementation of tables representing answers.

It is often the case that, in an answer for a sub-formula, the instantiated formulas are independent of the instantiations of some of the free variables. In this case, the table explicitly records only the partial instantiations that are needed. This method is described in [43] and [2] as tables with "hidden variables". In the tables representing answers to formulas, it is natural to have non-existence of a tuple correspond to associating the formula False

x	y	z	χ
1	*1*	*1*	$E(1,1) \wedge E(1,1)$
2	*2*	*2*	$E(2,2) \wedge E(2,2)$

x	y	χ
1	*1*	$E(1,1) \wedge E(1,1)$
2	*2*	$E(2,2) \wedge E(2,2)$

χ
$[E(1,1) \wedge E(1,1)] \vee [E(2,2) \wedge E(2,2)]$

x	*val*	χ
1	*2*	$\top$
2	*1*	$\top$

x	y	*val*	χ
1	*1*	*3*	$\top$
1	*2*	*4*	$\top$
2	*1*	*2*	$\top$
2	*2*	*3*	$\top$

Table 3.2: Answers to ϕ_1*,* ϕ_2*,* ϕ_3*,* t_1 *and* t_2

F		
x	y	χ
1	1	ϕ_1
2	3	$\top$
3	2	ϕ_2

(a) False table t_1

T		
x	y	χ
1	1	$\neg\phi_1$
2	3	$\bot$
3	2	$\neg\phi_2$

(b) True table t_2, which is the complement of the False table t_1

Table 3.3: Complement of a False table.

with the tuple. However, negating a sparse table with this convention produces a very dense table in which many tuples are associated with the formula True. To help keep tables sparse, we employ two kinds of tables, one in which absence of a tuple corresponds to False, and one where it corresponds to True. The formal semantics of the relational algebra operation on True/False tables are given in [2]. Next, we present how each relational algebra operation on True/False tables is implemented in Enfragmo.

Complement

The complement of a False (resp. True) table is the True (resp. False) table, with the same set of tuples as the original table, but with the formulas associated with the tuples negated. For example, the True table 3.3(b) is the complement of the False table 3.3(a). Note that each formula ϕ in False table 3.3(a) is not equal to $\bot$. Then the negation of the formula ϕ is not equal to $\top$, and thus should appear in the True table 3.3(b). Furthermore, all the tuples not in the False table 3.3(a) are associated with formula $\bot$, and thus the tuples will be associated with the formula $\neg\bot = \top$, and should not be included in the True table 3.3(b).

Join

1. **Join of two False tables.**

   Join of two False tables T_1 with variables v_1, and T_2 with variables v_2, is the False table T with the set of variables $v_1 \cup v_2$. Recall that a True/False table is an extended table with each tuple t in the table also associated with a formula ϕ. The tuples of the joined table T are determined as follows. For each $< t_1, \phi_1 >$ in T_1 and $< t_2, \phi_2 >$ in T_2 such that t_1 and t_2 have the same values for the set of common variables $v_1 \cap v_2$,

F		
x	y	χ
1	1	ϕ_1
2	2	ϕ_2
2	3	$\top$

(a) False table t_1

F		
x	z	χ
1	2	ψ_1
2	1	$\top$
3	3	ψ_2

(b) False table t_2

F			
x	y	z	χ
1	1	2	$\phi_1 \wedge \psi_1$
2	2	1	ϕ_2
2	3	1	$\top$

(c) False table $t = t_1 \bowtie t_2$

Table 3.4: A False table t from joining two False tables t_1 and t_2.

F		
x	y	χ
1	1	ϕ_1
2	3	$\top$
3	2	ϕ_2

(a) False table t_1

T		
x	z	χ
1	2	ψ_1
2	1	$\bot$
3	3	ψ_2

(b) True table t_2

F			
x	y	z	χ
1	1	1	ϕ_1
1	1	2	$\phi_1 \wedge \psi_1$
1	1	3	ϕ_1
2	3	2	$\top$
2	3	3	$\top$
3	2	1	ϕ_2
3	2	2	ϕ_2
3	2	3	$\phi_2 \wedge \psi_2$

(c) False table $t = t_1 \bowtie t_2$

Table 3.5: A False table t from joining a False table t_1 with a True table t_2.

we insert $< t, \phi_1 \wedge \phi_2 >$ to T, where the values in the tuple t coincides with the values in t_1 and t_2. This can be implemented in linear time if the two tables are sorted based on the values of common variables. Table 3.4 shows an example of joining two False tables to obtain a False table. Note that if the result False table is very dense and have lots of True formulas, we can get a sparse True table by computing its complement.

2. **Join of a False table and a True table.**

   The join of a False table and a True table is similar to the join of two False tables, except that we also need to consider the tuples not in the True table, which are associated with the formula True. The algorithm for joining a False table with a True table is given in Algorithm 1. Table 3.5 shows an example of joining two different tables to obtain a False table. The domain of all variables is $\{1, 2, 3\}$.

Algorithm 1: Algorithm for joining a False table with a True table

input : False tables T_1 and True table T_2
output: False table $T = T_1 \bowtie T_2$
begin
Let v_1 and v_2 be the variables of T_1 and T_2, respectively ;
$v = v_1 \cup v_2$, $v_c = v_1 \cap v_2$, $T =$ empty True table with variables v ;
Sort T_1 and T_2 based on the values of the common variables in v_c ;
LRowInd = RRowInd = 0 ;
while $LRowInd < T_1.RowCount$ **and** $RRowInd < T_2.RowCount$ **do**
$< t_1, \phi_1 >= T_1.GetRow(LRowInd)$, $< t_2, \phi_2 >= T_2.GetRow(RRowInd)$;
Let the tuples tc_1 and tc_2 be the values of the common variables in v_c in t_1 and t_2, respectively, in some fixed order ;
if $tc_1 < tc_2$ **then**
for *each tuple t' sharing the same values for variable v_c as tc_1 not in T_2* **do**
Let the tuple t be the values of variables v from t_1 and t' ;
Insert $< t, \phi_1 >$ to T ;
LRowInd = LRowInd + 1 ;
else if $tc_1 > tc_2$ **then**
RRowEndInd = the index of the last tuple in T_2 holding the sames values for the common variables v_c as tc_2;
RRowInd = RRowEndInd + 1 ;
else
LRowEndInd = the index of the last tuple in T_1 holding the sames values for the common variables v_c as tc_1;
$tend_1 = T_1.GetRow(LRowEndInd).GetTuple()$;
RRowEndInd = the index of the last tuple in T_2 holding the sames values for the common variables v_c as tc_2;
for *each $< t_l, \phi_l >$ in T_1 between t_1 and $tend_1$* **do**
for *each t_r sharing the same values for variable v_c as t_l not in T_2* **do**
Let the tuple t be the values of variables v from t_l and t_r ;
Insert $< t, \phi_l >$ to T ;
for *each $< t_l, \phi_l >$ in T_1 between t_1 and $tend_1$* **do**
for *each $< t_r, \phi_r >$ in T_2 between t_2 and $tend_2$* **do**
Let the tuple t be the values of variables v from t_l and t_r ;
Insert $< t, \phi_l \wedge \phi_r >$ to T ;
LRowInd = LRowEndInd + 1, RRowInd = RRowEndInd + 1 ;
while $LRowInd < T_1.RowCount$ **do**
$< t_1, \phi_1 >= T_1.GetRow(LRowInd)$;
for *each tuple t' sharing the same values for variable v_c as t_1 that is not in T_2* **do**
Let the tuple t be the values of variables v from t_1 and t' ;
Insert $< t, \phi_1 >$ to T ;
LRowInd = LRowInd + 1 ;
return T ;

T		
x	y	χ
1	1	ϕ
2	2	$\perp$

(a) True table t_1

T		
x	z	χ
1	2	ψ

(b) True table t_2

T			
x	y	z	χ
1	1	1	ϕ
1	2	2	ψ
1	1	2	$\phi \wedge \psi$
2	2	1	$\perp$
2	2	2	$\perp$

(c) True table $t = t_1 \bowtie t_2$

Table 3.6: A True table t from joining two True tables t_1 and t_2.

3. **Join of two True tables.** The join of two True tables is similar to the join of two False tables, except that we also need to consider the tuples not in the True tables, which are associated with the formula True. The algorithm for joining two True tables is given in Algorithm 2. Table 3.6 shows an example of joining two True tables to obtain a True table. The domain of all variables is $\{1, 2\}$.

Union

The Union operation is implemented dually to the Join operation.

Division

The algorithm for performing the division operation on True/False tables is given in Algorithm 3. Table 3.7 shows an example of dividing a True table with variables $\{x, y\}$ using the set of variables $\{x\}$, to obtain a True table. The domain of all variables is $\{1, 2, 3\}$.

Projection

The Projection operation is implemented dually to the Division operation.

An alternative algorithm for those operations is also implemented. In the algorithm, instead of sorting the tables based on the values of the common variables, we construct a hash table from the left table with the keys being the tuple of values for the common variables. Then we loop through the second table, and make queries to the hash table to get the subtable of the left table with the values of common variables in its tuples coincide with the ones in the current tuple in the right table.

Algorithm 2: Algorithm for joining two True tables

```
input  : True tables T1 and T2
output: True table T = T1 ⋈ T2
begin
    Let v1 and v2 be the variables of T1 and T2, respectively ;
    v = v1 ∪ v2, vc = v1 ∩ v2, T = empty True table with variables v ;
    Sort T1 and T2 based on the values of the common variables in vc ;
    LRowInd = RRowInd = 0 ;
    while LRowInd < T1.RowCount and RRowInd < T2.RowCount do
        <t1, φ1> = T1.GetRow(LRowInd), <t2, φ2> = T2.GetRow(RRowInd) ;
        Let the tuples tc1 and tc2 be the values of the common variables in vc in t1
        and t2, respectively, in some fixed order ;
        if tc1 < tc2 then
            for each tuple t' sharing the same values for variable vc as tc1 not in T2 do
                Let the tuple t be the values of variables v from t1 and t' ;
                Insert <t, φ1> to T ;
            LRowInd = LRowInd + 1 ;
        else if tc1 > tc2 then
            for each tuple t' sharing the same values for variable vc as tc2 not in T1 do
                Let the tuple t be the values of variables v from t2 and t' ;
                Insert <t, φ2> to T ;
            RRowInd = RRowInd + 1 ;
        else
            LRowEndInd = the index of the last tuple in T1 holding the sames values
            for the common variables vc as tc1 ;
            tend1 = T1.GetRow(LRowEndInd).GetTuple() ;
            RRowEndInd = the index of the last tuple in T2 holding the sames values
            for the common variables vc as tc2 ;
            tend2 = T2.GetRow(RRowEndInd).GetTuple() ;
            for each <tl, φl> in T1 between t1 and tend1 do
                for each tr sharing the same values for variable vc as tl not in T2 do
                    Let the tuple t be the values of variables v from tl and tr ;
                    Insert <t, φl> to T ;
            for each <tr, φr> in T2 between t2 and tend2 do
                for each tl sharing the same values for variable vc as tr not in T1 do
                    Let the tuple t be the values of variables v from tl and tr ;
                    Insert <t, φr> to T ;
            for each <tl, φl> in T1 between t1 and tend1 do
                for each <tr, φr> in T2 between t2 and tend2 do
                    Let the t be the values of variables v from tl and tr ;
                    Insert <t, φl ∧ φr> to T ;
            LRowInd = LRowEndInd + 1, RRowInd = RRowEndInd + 1 ;
    process remaining rows in a similar way to the first two cases in above while loop ;
    return T ;
```

T		
x	y	χ
1	1	ϕ_1
1	3	ϕ_2
2	1	ϕ_3
3	1	ϕ_4
3	3	ϕ_5

(a) True table t

T	
y	χ
1	$\phi_1 \wedge \phi_3 \wedge \phi_4$
3	$\phi_2 \wedge \phi_5$

(b) True table $t' = d_{\{x\}}(t)$

Table 3.7: A True table t' from dividing a True table t using variables {x}.

Algorithm 3: Algorithm for dividing a True/False table

input : True/False table T with variables V, and a set of variables $D \subseteq V$
output: True/False table $T' = d_D(T)$
begin
    create an empty table T' of the same kind as T with the variables $V \setminus D$;
    for *each possible instantiation t of the set of variables $V \setminus D$* **do**
        Let R be the set of tuples in T sharing the same values for the variables $V \setminus D$ as t ;
        Let Ψ be the set of formulas associated to the tuples in R, with the variables in $V \setminus D$ instantiated by the assignments in t ;
        if *T is a True table* **or** *T is a False table and contains all possible instantiations of variables in D* **then**
            $\phi = \bigwedge_{\psi \in \Psi} \psi$;
            Insert $< t, \phi >$ to T' ;
    return T' ;

A simple term is a term whose denotation can be computed (with respect to an instance), just using the assignments to its free variables. For example, $W(l)$ in Example 4 is a simple term. The formula for each tuple in an answer to a simple term is either True or False. A term which is not simple is called complex. The Count aggregate used in Example 4 is a complex term, because its value depends on the expansion predicate Q (the solution). To represent the answer to a complex term occurring as an argument to sub-formula ϕ, we have two data structures: 1) A hash-map which maps each value o which term t may denote to a table which is an answer to the formula $t(\bar{x}) = o$, and 2) A table which can be viewed as being the answer to the formula $\phi(\bar{x}, y) : t(\bar{x}) = y$. Methods for efficiently constructing the answers to complex terms, such as terms containing nested count and sum aggregates, are examined in [5] and [3].

3.4.3 CNF Transformation

The set of answers for the sentences of a specification are then transformed to a propositional CNF formula. As usual, this is done using a refinement of Tseitin's polytime transformation to CNF [62]. The major refinement is to rewrite the formula into negation normal form so that negations occur only on atoms. We also flatten nested conjunctions and disjunctions, and merge identical sub-formulas.

3.5 Experimental Evaluation

In this section, we compare the performance of Enfragmo to other grounding-based systems. A set of NP-hard of problems were chosen from [18]. We excluded problems for which all instances in the collection are easy. We also excluded problems where the sum aggregate is central, as the current implementation of sum in Enfragmo is preliminary and does not perform well. The other solvers are Clingo (v 3.0.3) [27], DLV (v 2010-10-14) [15], and IDP (v 2.20) [68]. Note that the input languages of the above systems are able to describe all the problems in NP. One of the dificulties in comparing performance of these systems is that the perfomance may be very sensitive to the problem specifications used, i.e., different specifications for the same problem may result in different performance in a single solver. In this experiment, for each system, we used specifications provided by the system authors, obtained from [18]. The experiments were run on an Intel Xeon L5420 quad-core 2.5 GHz processor, with a timeout of 600 seconds. All specifications, instances, solution verifiers,

and scripts used for the experiments can be downloaded from [1]. The results are given in Table 3.8. The entry n/t indicates that n instances were solved, each within the 600 second timeout, in a total time of t seconds. The time t includes the time for all the runs that timed out.

Table 3.8: Performance comparison of Enfragmo and other systems.

Problem	# Inst	Clingo	DLV	IDP	Enfragmo
GraphColoring	29	9/12400	8/13398	9/12199	**27/6965**
HamiltonianPath	29	**29/1.6**	20/6856	29/2.1	29/308
SchurNumbers	29	29/889	18/8273	28/1452	**29/643**
BlockedNQueens	29	**29/165**	28/9870	29/896	29/1278
ConnectedDominatingSet	20	**20/969**	13/6190	17/3258	19/2038
DisjunctiveScheduling	10	10/1174	5/3581	10/1008	**10/421**
Total	146	126/15601	92/48170	122/18818	**143/11656**

Table 3.8 shows that Enfragmo was able to solve almost all the instances in the collection, and performed the best on three of the six problems. Enfragmo also performed the best by the aggregate measures of total number of instances solved and total time spent.

3.6 Related Work

One framework for solving search problems is the *model expansion* framework, where we are given an instance of a problem and search for a solution satisfying certain properties. The framework is considered suitable representing search problems because it is computationally simpler task compared to satisfiability, e.g., satisfiability problem for FO is undecidable while model expansion, on finite domains, is in NP. One well-studied approach to solve model expansion tasks is to transform the high level problem specification and instance description into equivalent description in low level language, such as SAT, ILP, and CSP, for which we have efficient solvers available. In this section, we discuss what other people have done to solve model expansion tasks using this approach.

MXG [43, 48] is a framework based on classical first-order logic extended with inductive

definition. The set of the problems that can be described using MXG language is exactly those problems in NP. It is developed explicitly based on the model expansion framework. The MXG grounder translates its inputs into a propositional formula (a SAT instance), and the resulting SAT instance can be input to any off-the-shelf SAT solver. Similar to Enfragmo, the grounding in MXG is done in a bottom-up manner, based on algebraic database theory. Thus almost all the optimization techniques developed in the database community can be applied during the MXG grounding phase. MXG grounder works well with relatively small size tables. It can perform very well in lots of cases since instance predicates are normally sparse. However, when the tables for the instance predicates are negated, they result in very large tables. This issue is handled in Enfragmo by using the concept of True/False tables. The concept of hidden variables is also used in Enfragmo grounder to avoid instantiating unnecessary variables. MXG input language does not have built-in arithmetic functions and aggregate, which makes some interesting real-world problems not be conveniently expressed in MXG input language. Examples of such problems include Knapsack, Scheduling, and other problems involving weights and costs.

KodKod [61, 60] is another SAT-based constraint solver for an extension of first-order logic. Its grounder also translates the program input into a pure SAT instance in a bottom-up manner, then uses a SAT solver to search for a solution. Its grounding procedure is based on *Sparse-matrix representation of relations*, where each relation over a finite universe is represented as a matrix of Boolean values. Details can be found from [60]. The grounding approach used in KodKod and MXG, though they look quite different on the surfice, have close correspondence. The table used in MXG can be seen as a data structure representing the sparse matrix. On the other hand, every data structure used in KodKod to represent the matrix can also be used to represent the answer to a formula in MXG. One difference between the two grounders is that MXG translates its input into a ground formula, before it gets further transformed into CNF, while KodKod directly translates its inputs into equivalent CNF by assigning a CNF variable to each subformula. By design, the number of variables in KodKod grounding output is generally larger than that in MXG's. In some cases, manipulating the collection of possible values for relation variables in KodKod results in huge non-sparse matrices, which affects the performance of the KodKod system.

The IDP system [68, 69, 67, 38] is a model finder for the first-order logic extended with inductive definition, arithmetic functions and aggregates. The input language of IDP system is very similar to that of Enfragmo. The IDP system supports full inductive definition of

the ID logic FO(ID) [17], while Enfragmo only supports a special case. The grounder of the IDP translates its input into the language of its own low level solver, MinisatID. MinisatID extends the classical Minisat SAT solver [19] by adding supports of the inductive definition, arithmetic and aggregate functions. IDP's grounder takes a top-down approach. It first computes the *true bound* and the *false bound* from its inputs by representing the bounds symbolically as a first-order formula of instance predicates. The main advantage of symbolic computation is that the performance of the computation is independent of the size of variables' domains. On the other hand, in the symbolic computation of bounds, one needs to have access to a function which can decide whether two given FO formulas are equivalent. We know that this problem is generally undecidable. However, in the IDP grounder, the formulas are converted into a canonical format represented in first-order Binary Decision Diagram (BDD) structure, and two formulas having the same canonical representations are considered the same. It is claimed that a first-order BDD with 12 decision nodes is sufficient to handle most of real world specifications [67]. The performance of the IDP system deeply relies on the quality of computed bounds. On the other hand, the bound computation phase will terminate after a fixed amount of time if not completed. Thus, it is not intended to find the best bounds possible. One can construct specifications for which IDP fails to find complete bounds. In [63], the authors show how a lifted version of Unit Propagation technique can be used in Enfragmo system to always compute the complete bounds by, instead of symbolic computation, using information from the instance structure. Another potential disadvantage of IDP system comes from the fact that IDP grounder translates its input into a special syntax only accepted by its specialized solver. The output of the IDP grounder is not pure CNF, thus is not accepted by normal SAT solvers. In order for the IDP system to benefit from the latest techniques, the existing low level solver (MinisatID) has to be extended using corresponding new techniques.

Answer Set Programming (ASP) is a framework for declarative problem solving based on the stable model semantics [32]. The existing high level ASP solvers are, e.g., Clingo [27, 28] and DLV [15, 49]. ASP systems work by grounding the input logic program by substituting the variables by ground terms in the Herbrand universe in every possible way. The instantiated program is then solved by propositional logic programming solvers. The difference between languages of Enfragmo and ASP systems is that Enfragmo allows arbitrary sentences in an extended FO and inductive definition, while the ASP program is one big inductive definition under the stable model semantics, and in the definition, each rule

body is restricted to conjunctions of literals and weight constraints.

3.7 Conclusion

We presented the Enfragmo system for modelling and solving combinatorial search problems. It provides users with a convenient way to specify and solve computationally hard problems, in particular search problems whose decision versions are in the complexity class NP. The performance of the Enfragmo system is comparable to that of related systems. As future work, we plan to optimize the implementation of the inductive definition solver and Sum aggregates, develop more optimization techniques in our grounder, and extend the grounder to ground to other low level languages like ILP or SMT.

Chapter 4

Solving Modular Model Expansion

4.1 Introduction

The research described in this chapter is a part of a research program of developing formal foundations for specification/modelling languages (declarative programming) for solving computationally hard problems. Mitchell and Ternovska [42] formalize search problems as the logical task of *model expansion (MX)*, the task of expanding a given (mathematical) structure with new relations. They started a research program of finding common underlying principles of various approaches to specifying and solving search problems, finding appropriate mathematical abstractions, and investigating complexity-theoretic and expressiveness issues. The next step in the development of the MX-based framework is adding modularity concepts. The following example clarifies our goals.

Example 11 (Business Process Planner) *Business Process Planner takes a set S of services and a set R of restrictions (such as dependencies between services or their deadlines) and generates a plan P in the output. Each $Provider_i$ is allocated a potential subset of services S_i and restrictions R_i on it. The provider then generates a set of potential plans P_i (and their associated costs) and returns it to the planner. Depending on whether the planner is satisfied with the partial plans, it may reconsider service allocations or relax restrictions on services. The output plan P is generated by combining plans P_i. The business process planner relies on external services for particular tasks. The tasks performed by each of the providers or the planner are often NP-complete, e.g. the Traveling Salesman Problem. Therefore, finding a combined solution is a computationally (as well as conceptually) complex*

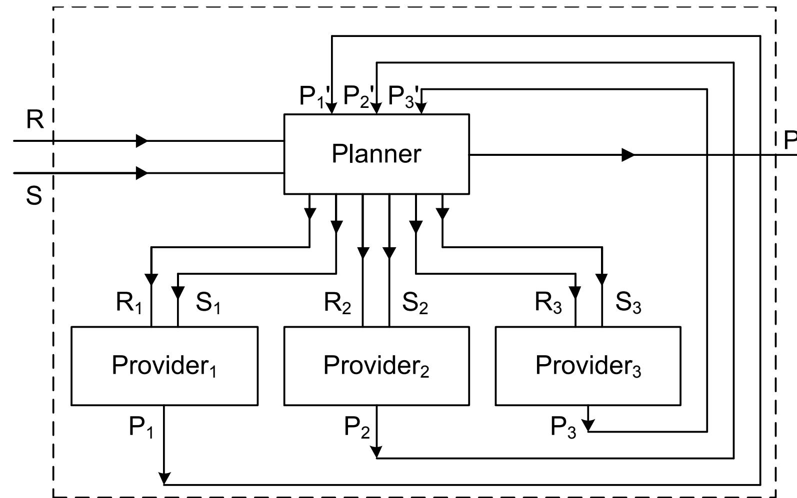

Figure 4.1: Business Process Planner

task. Such a central planner could be used in business process management in many areas. We give three examples below.

Logistics Service Provider *is a high level approach to use contracted carriers, local post, fleet management, driver dispatch, warehouse services, transportation management systems, e-business services as well as local logistics service providers with their own sub-modules.*

Manufacturer Supply Chain Management *uses a supply chains planner relying on transportation, shipping services, various providers for inventory spaces, etc.. It uses services of third party logistics (3PL) providers, which themselves depend on services provided by smaller local companies.*

Mid-size Businesses Relying on External Web Services and Cloud Computing *Such businesses often use data analysis services, storing, spreadsheet software (office suite), etc.. The new cloud-based software paradigm satisfies the same need in the domain of software systems.*

We would like to find a method for finding solutions to such complex tasks. Modularity is incorporated through representing each part in the most suitable language. For example, the planner module is more easily specified in extended first-order logic, while some provider

modules perhaps are most easily specified using MILP (mixed integer linear programming). Note that all the applications mentioned in the example are increasingly web-based, and declarative programming of a web service is an open problem. In this thesis and related papers, we take initial steps towards solving some aspects of this problem, namely the underlying computationally complex task.

Such a method should treat each primitive module as a black-box (i.e., should not assume access to a complete axiomatization of the module). Not assuming complete knowledge is essential in solving problems like business process planning. This is because each of the solid boxes in Figure 4.1 represents a business entity which often, while interested in participating in the process, is not necessarily willing to share the information that has affected their decisions. Therefore, any approach to representing and solving such systems that assumes unlimited access to complete axiomatizations of these entities is impractical.

In recent work [53], the authors extended the MX framework to be able to represent a modular system. The most interesting aspect of that proposal is that modules can be considered from both model-theoretic and operational views. Under the model-theoretic view, an MX module is a set (or class) of structures, and under the operational view it is an operator, mapping a subset of the vocabulary to another subset. An abstract algebra on MX modules is given, and it allows one to combine modules on an abstract model-theoretic level, independently from what languages are used for describing them. Perhaps the most important operation in the algebra is the loop (or feedback) operation, since iteration underlies many solving methods. The authors show that the power of the loop operator is such that the combined modular system can capture all of the complexity class NP even when each module is deterministic and polytime. Moreover, in general, adding loops gives an increase in the polynomial time hierarchy, one step from the highest complexity of the components. It is also shown that each module can be viewed as an operator, and when each module is (anti-) monotone, the number of potential solutions can be significantly reduced by using ideas from the logic programming community.

To develop the framework further, we need a method for "solving" modular MX systems. By solving we mean finding structures which are in the modular system, where the system is viewed as a function of individual modules. The goal is to come up with a way to find the structures in the given modular system. Since we aim at developing the foundations of language-independent problem solving, we tackle the problem model-theoretically.

We take our inspiration in how "combined" solvers are constructed in the general field of declarative problem solving. The field consists of many areas such as Integer Linear Programming (ILP), Answer Set Programming (ASP), Satisfiability Modulo Theories (SMT), Satisfiability (SAT), and Constraint Programming (CP), and each of these areas has developed multitudes of solvers, including powerful "combined" solvers such as SMT solvers. Moreover, SMT-like techniques are needed in the ASP community [44]. Our main challenge is to come up with an appropriate mathematical abstraction of "combined" solving. Note that existing "combined" solvers are very powerful, but in some sense are not general enough to solve arbitrary modular systems. They are designed to solve problems axiomatized in their designated languages, e.g., SMT solvers for SAT extended with different theories, Constraint Answer Set solvers for the combination of ASP and CP language, etc. *Our goal of this paper is to design an algorithm which takes an arbitrary modular system as input and finds a structure in the modular system.* Note that our hope is not to compete with existing systems, nor to replace them, but to build something completely new, that contributes to solving computationally complex tasks that are presented in a modular way and interact with each other during solving. In order to achieve this general goal, we need to study existing systems for combing solvers.

Our contributions of this chapter are listed below:

1. We formalize common principles of "combined" solving in different communities in the context of modular model expansion. Just as in [53], we use a combination of a model-theoretic, algebraic and operational view of modular systems.

2. We design an abstract algorithm that, given a modular system, computes the models of that modular system iteratively, and we formulate conditions on languages of individual modules to participate in the iterative solving. Correctness of our algorithm is proven model-theoretically.

3. We introduce abstractions for many ideas in practical systems such as the concept of a *valid acceptance procedure* that abstractly represents unit propagation in SAT, well-founded model computation in ASP, arc-consistency checkers in CP, etc.

4. We show that, in the context of the model expansion task, our algorithm generalizes the work of solvers from different communities in a unifying and abstract way. In particular, we show that DPLL(T) framework [45], branch-and-cut based ILP solver

[47] and state-of-the-art combination of ASP and CP [30] are all specializations of our algorithm.

5. For each system (DPLL(T), ILP, ASP and CP) with a problem specification, we design a compound modular system which takes a problem instance as input, and outputs the solution, such that the set of structures in the modular system corresponds to the set of solutions for the given problem specification. For example, for the DPLL(T) system with a specification ϕ which axiomatizes a scheduling problem, we design a compound modular system such that the set of the structures in the modular system represents all pairs of possible scheduling instances and corresponding solutions (schedule).

6. Each primitive modules in a modular system could be described in different language, e.g., a modular system constructed from joining two primitive modules, where one module is described in the language of ASP, and the other one is axiomatized using the CP language. We show how our algorithm can benefit from the techniques used in practical solver constructions to solve the modular system described in different language efficiently. For example, we show that the unit propagation techniques in ASP solver construction, and the constraint propagation techniques in the CP solver implementation can be used to solve the modular system described in both languages efficiently.

7. For each existing system S (DPLL(T), ILP, and ASP and CP), we construct a modular system M such that our algorithm on M models the solving procedure of the corresponding system S. In this way, we show the feasibility of our algorithm for solving arbitrary modular systems.

4.1.1 My Contributions

The work described in this chapter is based on the joint work with Shahab Tasharrofi [58, 56]. Shahab contributed the formal definitions of the modular systems and the operations to combine the modules. The algorithm for computing the models of modular systems is jointly designed. The work on modelling existing frameworks is done mainly by me with some help from Shahab Tasharrofi.

4.2 Background

4.2.1 Modular Systems

This section reviews the concept of a modular system defined in [53] based on the initial development in [36]. As in [53], *each modular system abstractly represents an MX task*, i.e., a set (or class) of structures over some instance (input) and expansion (output) vocabulary. A modular system is formally described as a set of primitive modules (individual MX tasks) combined using the operations of: (1) Projection($\pi_\tau(M)$) which restricts the vocabulary of a module, (2) Composition($M_1 \rhd M_2$) which connects outputs of M_1 to inputs of M_2, (3) Union($M_1 \cup M_2$), (4) Feedback($M[R = S]$) which connects output S of M to its inputs R and, (5) Intersection($M_1 \cap M_2$). In this chapter, we only consider modular systems which do not use the union operator.

Formal definitions of these operations are not essential for understanding this chapter, thus, we refer the reader to [53] for details. We illustrate these operations by giving the following algebraic specification for the modular system in Example 11.

$$BPP := \pi_{\{R,S,P\}}(Planner \rhd (Provider_1 \cap Provider_2 \cap Provider_3)) \\ [P_1 = {P_1}'][P_2 = {P_2}'][P_3 = {P_3}']. \tag{4.1}$$

Considering Figure 4.1, symbol BPP refers to the whole modular system denoted by the box with dotted borders. R, S and P are the only vocabulary symbols important outside BPP. So, other symbols are projected out. Also, there are three feedbacks from P_i ($i \in \{1, 2, 3\}$) to P_i'. Since each modular system is a set of structures, each such structure is called a *model* of that system. We are looking for models of a modular system M which expand a given instance structure $\mathcal{A}$. Those are *M's solutions for $\mathcal{A}$*.

Our goal is to give a method to solve the MX task for a given modular system, i.e., given a modular system M and structure $\mathcal{A}$, find $\mathcal{B}$ in M which expands $\mathcal{A}$. We find our inspiration in existing solver architectures by viewing them at a high level of abstraction.

4.3 Computing Models of Modular Systems

In this section, we introduce an algorithm which takes a modular system M and a structure $\mathcal{A}$ and finds an expansion $\mathcal{B}$ of $\mathcal{A}$ in M. Our algorithm uses a tool external to the modular system (a solver). It uses modules of a modular system to "assist" the solver in finding a

model (if one exists). Starting from an empty expansion of $\mathcal{A}$ (i.e., a partial structure which contains no information about the expansion predicates), the solver gradually extends the current structure (through an interaction with the modules of the given modular system) until it either finds a model that satisfies the modular system or concludes that none exists. To model this procedure, we use the notion of the partial structure defined in 3.2.3.

In the following, by structure we always mean a total structure, unless otherwise specified. We may talk about "bad" partial structures which, informally, are the ones that cannot be extended to a structure in M. Also, when we talk about a τ_p-partial structure, in the MX context, τ_p is always a subset of ε.

Total structures are partial structures with no unknown values. Thus, in the algorithmic sense, total structures need no further guessing and should only be checked against the modular system. A good algorithm rejects "bad" partial structures sooner, i.e., the sooner a "bad" partial structure is detected, the faster the algorithm is.

Up to now, we defined partial and total structures and talked about modules rejecting "bad" partial structures. However, modules are sets of structures (in contrast with sets of partial structures). Thus, acceptance of a partial structure has to be defined properly. Towards this goal, we first formalize the informal concept of "good" partial structures. The actual acceptance procedure for partial structures is defined later in the section.

Definition 5 (Good Partial Structures) *For a set of structures S and partial structure $\mathcal{B}$, we say $\mathcal{B}$ is a* good partial structure wrt *S if there is $\mathcal{B}' \in S$ which extends $\mathcal{B}$.*

4.3.1 Requirements on the Modules

As expressed in the introduction, there is practical desire to solve complex computational tasks in a modular way so that full access to a complete axiomatization of the module is not assumed, i.e., the module is treated as a black box and accessed via controlled methods. However, clearly, as the solver does not have any information about the internals of the modules, it needs to be assisted by the modules themselves. Therefore, the next question could be: "what assistance does the solver need from modules so that its correctness is always guaranteed, i.e., the solver only returns correct solutions (structures in the modular system)?" Intuitively, modules should be able to tell whether the solver is on the "right" direction or not, i.e., whether the current partial structure is bad, and if so, tell the solver to stop developing this direction further. We accomplish this goal by letting a module accept

or reject a partial structure produced by the solver and, in the case of rejection, provide a "reason" to prevent the solver from producing the same model later on. Furthermore, a module may "know" some extra information that a solver does not. Due to this fact, modules may give the solver some hints to accelerate the computation in the current direction. Our algorithm models such hints using "advice" to the solver.

Definition 6 (Advice) *Let Pre and Post be formulas in a language $\mathcal{L}$. Formula ϕ is by definition $Pre \supset Post$, which is* advice *wrt a partial structure $\mathcal{B}$ and a set of structures M if:*

1. $\mathcal{B} \models Pre$,
2. $\mathcal{B} \not\models Post$ *and,*
3. *for every total structure $\mathcal{B}'$ in M, we have $\mathcal{B}' \models \phi$.*

The role of advice is to prune the search and to accelerate extending a partial structure $\mathcal{B}$ by giving a formula that is not yet satisfied by $\mathcal{B}$, but is satisfied by any total extensions of $\mathcal{B}$ in M. *Pre* corresponds to the part that is satisfied by $\mathcal{B}$ and *Post* corresponds to the unknown part that is not yet satisfied by $\mathcal{B}$.

Note that in order to pass advice to a solver, there should be a common language that the solver and the modules understand (although it may be different from all internal languages of the modules). Such a language should satisfy the following properties:

Definition 7 (Solver Language) *For a language $\mathcal{L}$, we say $\mathcal{L}$ is a* solver language *if:*

- *If ϕ is a ground atom (i.e., $R(t_1, \cdots, t_n)$ in language $\mathcal{L}$ where R is an n-ary predicate symbol and $t_1, \cdots, t_n$ are variable-free terms in language $\mathcal{L}$), then $\phi \in \mathcal{L}$. Also, if $\phi_1, \phi_2 \in \mathcal{L}$ then $\neg\phi_1 \in \mathcal{L}$ and $(\phi_1 \supset \phi_2) \in \mathcal{L}$.*
- *Satisfiability relation for $\mathcal{L}$ respects the standard extension of FO satisfiability relation to partial structures.*
- *Satisfiability relation for $\mathcal{L}$ gives a classical semantics to connectives $\neg$ (negation) and $\supset$ (implication).*
- *$\mathcal{L}$ is monotone, i.e., for sets of axioms Γ, Γ': $\Gamma \subseteq \Gamma' \Rightarrow Con_L(\Gamma) \subseteq Con_L(\Gamma')$.*

Note that we are defining a family of languages. Except for the first item in the definition, the other items are not part of the language itself. They are the properties of the language should have. Also note that the third item in the definition of the solver language implies the resolution theorem, i.e., $\Gamma \models_L A \supset B$ implies $\Gamma \cup \{A\} \models_L B$, and the deduction theorem, i.e., $\Gamma \cup \{A\} \models_L B$ implies $\Gamma \models_L A \supset B$. The resolution theorem in Definition 7 guarantees that, once an advice of form $Pre \supset Post$ is added to the solver, and when the solver has deduced Pre under some assumptions, it can also deduce $Post$ under the same assumptions; while the deduction theorem allows the modules to generate the advice accordingly. From now on, we assume that our advice and reasons are expressed in a language as above, i.e., a solver language.

We talked about modules assisting the solver, but a module is a set of structures and has no computational power. Instead, we associate each module with an "oracle" to accept/reject a partial structure and give "reasons" and "advice" accordingly. Note that assuming access to oracles which accept a partial structure iff it is a good partial structure, one can always find a total model by polynomially many queries to such oracles. While theoretically possible, in practice, access to oracles with such a strong acceptance procedure is usually not provided, and most practical solvers apply propagation through more efficient and simple local consistency checking methods. Thus, we have to (carefully) relax our assumptions for a weaker procedure, which we call a Valid Acceptance Procedure.

Definition 8 (Valid Acceptance Procedure) *Let S be a set of τ-structures. We say that P is a* valid acceptance procedure *for S if for all τ_p-partial structures $\mathcal{B}$, we have:*

- *If $\mathcal{B}$ is total, then (1) P accepts $\mathcal{B}$ if $\mathcal{B} \in S$, and (2) P rejects $\mathcal{B}$ if $\mathcal{B} \notin S$.*
- *If $\mathcal{B}$ is not total but $\mathcal{B}$ is good wrt S, then P accepts $\mathcal{B}$.*
- *If $\mathcal{B}$ is neither total nor good wrt $\mathcal{B}$, then P is free to either accept or reject $\mathcal{B}$.*

The procedure above is called valid as it never rejects any good partial structures. However, it could be a weak acceptance procedure because it may accept some bad partial structures. This kind of weak acceptance procedure is abundant in practice, e.g., Unit Propagation in SAT, Arc-Consistency Checks in CP, and computation of Founded and Unfounded Sets in ASP. As these examples show, such weak notions of acceptance can usually be implemented efficiently as they only look for local inconsistencies. Informally, oracles accept/reject a partial structure through a valid acceptance procedure for a set containing all

possible instances of a problem and their solutions. We call this set a Certificate Set. Before giving its formal definition, we should however point out one difference to the readers who are not accustomed to the logical approach to complexity: In theoretical computer science, a problem is a subset of $\{0,1\}^*$. However, in descriptive complexity, the equivalent definition of a problem being a set of structures is adopted. Now, we give the formal definition of the Certificate Set.

Definition 9 (Certificate Set) *Let σ and ε be instance and expansion vocabularies, respectively. Let $\mathcal{P}$ be a problem, i.e., a set of σ-structures, and C be a set of $(\sigma \cup \varepsilon)$-structures. Then, C is a $(\sigma \cup \varepsilon)$-*certificate set for *$\mathcal{P}$ if for all σ-structures $\mathcal{A}$: $\mathcal{A} \in \mathcal{P}$ iff there is a structure $\mathcal{B} \in C$ that expands $\mathcal{A}$.*

Example 12 (Graph 3-coloring: Certificates) *Consider Example 1 of graph 3-coloring. There, $\sigma = \{E\}$ and $\varepsilon = \{R, G, B\}$. The problem P is the set of graphs $\mathcal{G} = (V^{\mathcal{G}}; E^{\mathcal{G}})$ which are 3-colorable. A certificate set C for problem P of graph 3-coloring is, as one might expect, the same as 3-coloring certificates in complexity theory, i.e., a partitioning of vertices into three sets R, G and B such that no edge of the graph connects vertices of the same color together. The certificate set C, as expected, should be such that $\mathcal{A} \in P$ (i.e., $\mathcal{A}$ is 3-colorable) iff C has at least one 3-coloring for $\mathcal{A}$ (i.e., there is at least one expansion $\mathcal{B}$ of $\mathcal{A}$ in C which interprets R, G and B correctly).*

Recall that each module is associated with an oracle to accept/reject a partial structure and give reasons and advice accordingly. The role of the reasons is to prevent some bad structures and their extensions from being proposed more than once, i.e., when a model is deduced to be bad by an oracle, a new reason is provided by the oracle and added to the solver such that all models of the system satisfy that reason but the "bad" structure does not. The role of advice is to provide useful information to the solver (satisfied by all models) but not yet satisfied by the partial structure $\mathcal{B}$. Next, we present conditions that oracles should satisfy so that their corresponding modules can contribute to our algorithm.

Definition 10 (Oracle Properties) *Let $\mathcal{L}$ be a solver language. Let $\mathcal{P}$ be a problem, and let O be an oracle. We say that O is:*

- Complete and Constructive (CC) wrt $\mathcal{L}$ *if O returns a reason $\psi_{\mathcal{B}}$ in $\mathcal{L}$ for each partial structure $\mathcal{B}$ that it rejects such that: (1) $\mathcal{B} \models \neg\psi_{\mathcal{B}}$ and, (2) all total structures accepted by O satisfy $\psi_{\mathcal{B}}$.*

- Advising (A) wrt $\mathcal{L}$ *if* O *gives a (possibly empty) set of advices in* $\mathcal{L}$ *wrt* $\mathcal{B}$ *for all partial structure* $\mathcal{B}$.
- Verifying (V) *if* O *is a valid acceptance procedure for some certificate set* C *for* P.

Oracle O differs from the usual oracles in the sense that it not only gives yes/no answers, but also provides a reason for its "no" answers. It is *complete wrt* $\mathcal{L}$ because it ensures the existence of such a reason and *constructive* because it provides such a reason. Also, it is *advising* because it provides some facts that were previously unknown to guide the search. Finally, it is *verifying* because it guides the partial structure to a solution through a valid acceptance procedure. Although the procedure can be weak as described above, good partial structures are never rejected and O always accepts or rejects total structures correctly. This property guarantees the convergence to a total model. In the following sections, we use the term CCAV oracle to denote an oracle which is complete, constructive, advising, and verifying. Properties of CCAV oracles are later used in Proposition 1 to prove the correctness of our algorithm.

Example 13 (Graph 3-coloring: Reasons and Advice) *Consider the graph 3-coloring example of Example 1. We want to describe some possible scenarios for an oracle* O *of graph 3-coloring. Consider graph* $\mathcal{G} = (V^{\mathcal{G}}; E^{\mathcal{G}})$ *with* $V^{\mathcal{G}} = \{a, b, c, d\}$ *and* $E^{\mathcal{G}} = \{(a, b), (b, a), (a, c), (c, a), (a, d), (d, a), (c, d), (d, c)\}$. *Also consider a partial expansion* $\mathcal{B} = (V^{\mathcal{G}}; E^{\mathcal{G}}, R^{\mathcal{B}}, B^{\mathcal{B}}, G^{\mathcal{B}})$ *of* $\mathcal{G}$ *to* $\{R, B, G\}$ *which assigns color red to vertices* a *and* b*, color green to vertex* c *and (yet) no color to vertex* d. *Obviously,* $\mathcal{B}$ *is a bad partial 3-coloring and no matter what color we assign to* d*, we will not obtain a valid 3-coloring. Therefore, one scenario for oracle* O *is to reject this partial coloring and give a reason like:* $\neg(R(a) \wedge R(b))$.

However, oracles do not always recognize a bad partial structure right away (recall that although oracles are valid acceptance procedures, they can be weak). Therefore, another scenario for O *is to accept* $\mathcal{B}$ *but still help the solver by giving the advice* $\psi := (R(a) \wedge G(c)) \supset B(d)$. *Formula* ψ *helps the solver to infer that* $\mathcal{B}$ *cannot be extended to a valid 3-coloring by checking only one of* $\mathcal{B}$*'s three possible extensions. The worst scenario, however, is that* O *accepts* $\mathcal{B}$ *and does not give any advice. In this case, the solver has to check all colors for* d *before inferring that* $\mathcal{B}$ *is a bad partial structure.*

Implementation of Oracles

When a module is described using some well-studied language, we often have existing efficient Valid Acceptance Procedures used in solver constructions, e.g., Well-Founded Model computation for ASP, Arc-Consistency checking for CP, Theory Propagation for various SMT theories, a lifted version of Unit Propagation [63] for FO, etc. In these cases, corresponding techniques can be used to implement oracles to accept/reject partial structures and to provide reasons and advice accordingly. For example, we could use the propagation techniques used in the CP community to design an oracle for a primitive module described in some CP language. We may not simply use the CP solver to solve the whole problem since we may have other primitive modules axiomatized in other languages such as ASP or ILP. In this case, the propagation techniques in the ASP or ILP communities can also be used to construct oracles for the corresponding primitive modules which assist the solver to solve the modular system with each primitive modules described in different languages. In this way, we benefit from the techniques used in practical solver constructions in different communities to solve the modular system efficiently. More examples of Valid Acceptance Procedures used in practice are given in Section 4.4.

4.3.2 Requirements on the Solver

The role of the solver is to provide a possibly good partial structure to the oracles, and if none of the oracles reject the partial structure, keep extending it until we find a solution or conclude no extension exists. If the partial structure is rejected by some oracle, the solver gets a reason from that oracle for rejection and tries some other partial structure. The solver also gets advice from oracles to accelerate the search. In this section, we discuss properties that a solver must satisfy in order for it to participate in our iterative solving procedure. Although the solver can be realized by many practical systems, for them to work in an orderly fashion and for algorithm to converge to a solution fast, it has to satisfy certain properties. First, the solver has to be online since the oracles keep adding reasons and advice to it. Furthermore, to guarantee termination, the solver has to guarantee progress, which means it either produces a proper extension of the previous partial structure or, if not, the solver is guaranteed to never return any extension of that previous partial structure later on. Now, we give the requirements on the solver formally.

Definition 11 (Complete Online Solver) *A solver S is* complete and online *if the following conditions are satisfied by S:*

- *S supports the actions of initialization, adding sentences (reasons and advices from oracles), and reporting its state as either $\langle UNSAT \rangle$ or $\langle SAT, \mathcal{B} \rangle$.*
- *If S reports $\langle UNSAT \rangle$ then the set of sentences added to S are unsatisfiable over the domain A,*
- *If S reports $\langle SAT, \mathcal{B} \rangle$ then $\mathcal{B}$ does not falsify any of the sentences added to S,*
- *If S has reported $\langle SAT, \mathcal{B}_1 \rangle, \cdots, \langle SAT, \mathcal{B}_n \rangle$ and $1 \leq i < j \leq n$, then either $\mathcal{B}_j$ is a proper extension of $\mathcal{B}_i$ or, for all $k \geq j$, $\mathcal{B}_k$ does not extend $\mathcal{B}_i$.*

For finite structures, a solver as above is (1) Sound: it returns partial structures that at least do not falsify any of the axioms in solver language, and (2) Complete: it reports unsatisfiability only when unsatisfiability is detected and not when, for example, some heuristic has failed to find an answer or some time limit is reached. Proposition 1 gives the exact correspondence in this regard.

4.3.3 Lazy Model Expansion Algorithm

In this section, we present an iterative algorithm to solve model expansion tasks for modular systems. This algorithm takes an instance structure and a modular system (and its CCAV oracles) and integrates them with a complete online solver to solve a model expansion task in an iterative fashion. The algorithm works by accumulating reasons and advice from oracles and gradually converging to a solution to the problem.

Algorithm 4 presents the lazy model expansion algorithm for solving general modular systems. The word "lazy" comes from the SMT community which refers to the integration of the DPLL-style reasoning and theory specific propagation techniques. The main idea is to incrementally build the expansion structure to satisfy all the primitive modules in the compound modular system in the input. Note that this simple approach respects all the operators defined for the modular system except the union operator.

Proposition 1 (Correctness) *Algorithm 4 is sound and complete* [1] *for finite structures,*

[1] *It follows the idea of completeness for search algorithms.*

Algorithm 4: Lazy Model Expansion Algorithm

Data: Modular System M with each module M_i associated with a CCAV oracle O_i, input structure $\mathcal{A}$ and complete online solver S
Result: Structure $\mathcal{B}$ that expands $\mathcal{A}$ and is in M
begin
  Initialize the solver S using the empty expansion of $\mathcal{A}$;
  while *TRUE* **do**
    Let R be the state of S ;
    if $R = \langle UNSAT \rangle$ **then return** *Unsatisfiable* ;
    else if $R = \langle SAT, \mathcal{B} \rangle$ **then**
      Add all the advice from oracles wrt $\mathcal{B}$ to S ;
      if M *does not accept* $\mathcal{B}$ **then**
        Find a module M_i in M such that M_i does not accept $\mathcal{B}|_{vocab(M_i)}$;
        Add the reason given by oracle O_i to S ;
      else if $\mathcal{B}$ *is total* **then return** $\mathcal{B}$;

---

i.e., given a modular system M with CCAV oracles, a complete online solver S and a finite instance structure $\mathcal{A}$:

1. *If Algorithm 4 returns $\mathcal{B}$, then $\mathcal{B} \in M$,*

2. *If Algorithm 4 returns* "Unsatisfiable" *then none of structures $\mathcal{B} \in M$ expands $\mathcal{A}$.*

3. *Algorithm 4 always terminates.*

Proof: (1) If structure $\mathcal{B}$ is returned, it is total and it is accepted by all oracles. So, as all oracles are verifying, total structures are decided correctly. Therefore $\mathcal{B}|_{vocab(M_i)} \in M_i$ for all primitive modules M_i of M. Thus, $\mathcal{B} \in M$.
(2) If "*Unsatisfiable*" is returned, then the set of sentences added to the solver S are unsatisfiable (by properties of S). Also, by monotonicity of language $\mathcal{L}$ of the solver, no superset of such set of sentences is satisfiable. Also, as these sentences are only advice and reasons returned by oracles, they are true in every $\mathcal{B} \in M$ which expands $\mathcal{A}$ (by properties of CCAV oracles). Therefore, there is no $\mathcal{B} \in M$ which expands $\mathcal{A}$.
(3) By the property of complete online solvers, S can never report $\langle SAT, \mathcal{B} \rangle$ (for some $\mathcal{B}$) twice. Therefore, as there are only finitely many different partial expansions of finite structure $\mathcal{A}$, either there should be a point where S reports $\langle SAT, \mathcal{B} \rangle$ for some total structure

$\mathcal{B}$ accepted by all modules (which terminates the algorithm), or a time when S reports $\langle UNSAT \rangle$ (which also terminates the algorithm). ∎

Proposition 2 (Time Complexity) *For all modular systems M, there is $k \in \mathbb{N}$ s.t. for each finite structure $\mathcal{A}$, Algorithm 4 terminates after at most $3^{O(|dom(\mathcal{A})|^k)}$ calls to the solver.*

Proof: Let k be the maximum arity of the predicate symbols in the expansion vocabulary of M. Then, the proof follows the proof for Proposition 1 plus the fact that $3^{O(|dom(\mathcal{A})|^k)}$ different partial interpretations exist. ∎

4.4 Case Studies: Existing Frameworks

In this section, we describe algorithms from three different areas and show that they can be effectively modelled by our proposed algorithm in the context of model expansion. We show that our algorithm acts similarly to the state-of-the-art algorithms used in the areas of SMT, ASP, and ILP, when the right components are provided.

Notation 1 *We sometimes use a τ-structure $\mathcal{B}$ (which gives an interpretation to vocabulary τ) as the set of τ-atoms of $\mathcal{B}$. For example, when $\tau = \{R, S\}$ and $R^{\mathcal{B}} = \{(1,2)\}$ and $S^{\mathcal{B}} = \{(1,1),(2,2)\}$, then we may use $\mathcal{B}$ to represent the following set of atoms:*

$$\mathcal{B} = \{R(1,2), S(1,1), S(2,2)\}.$$

We may also use a partial interpretation as a set of true atoms in a similar fashion. Sometimes, we also use $\mathcal{B}$ to represent a formula, i.e., the conjunction of the atoms in the above set. The complement of a set is defined as usual, e.g., $R^{\mathcal{B}^c} = dom(\mathcal{B})^2 \setminus R^{\mathcal{B}}$. Negation of a set S of literals is also defined such that $l \in S$ if and only if $\neg l \in \neg S$.

4.4.1 Modelling DPLL(T)

The DPLL(T) [45] system is an abstract framework to model the lazy SMT approach. It is based on a general DPLL(X) engine, where X can be instantiated with a theory T solver. The DPLL(T) engine extends the Decide, UnitPropagate, Backjump, Fail and Restart actions of the classic DPLL framework with three new actions: (1) **TheoryPropagate** gives literals that are T-consequences of the current partial assignment, (2) T**-Learn** learns T-consistent clauses, and (3) T**-Forget** forgets some previous lemmas of theory solver.

To participate in the DPLL(T) solving architecture, a theory solver provides three operations: (1) taking literals that have been set true, (2) checking if setting these literals true is T-consistent and, if not, providing a subset of them that causes inconsistency, (3) identifying some currently undefined literals that are T-consequences of the current partial assignment and providing a justification for each. More details can be found in [45].

In this section, the the following MX task is used as a running example to show how Algorithm 4 models the DPLL(T) system.

Example 14 (Disjoint Scheduling) *Given a set of Tasks, $\{t_1, \cdots, t_n\}$ and a set of constraints, the goal is to find a schedule that satisfies all the constraints. Each task t_i has an earliest starting time $EST(t_i)$, a latest ending time $LET(t_i)$ and a length $L(t_i)$. There are also two predicates $P(t_i, t_j)$ and $D(t_i, t_j)$ which say, respectively, that task t_i should end before task t_j starts, and two tasks t_i and t_j cannot overlap. We are asked to find the function $S(t_i)$ for the start time which satisfies the conditions above. In this example, $\sigma = \{EST, LET, L, P, D\}$ and $\varepsilon = \{S\}$.*

We solve the disjoint scheduling problem in Example 14 using the DPLL(T) system with the theory T being the Theory of Difference Logic [45].

Example 15 (Disjoint Scheduling (Specification, Instance, Ground Program)) *We use following specification to represent the disjoint scheduling problem.*

$$\begin{aligned}
&\forall t\ (EST(t) \leq S(t)),\\
&\forall t\ (S(t) + L(t) \leq LET(t)),\\
&\forall t_1 \forall t_2\ (P(t_1, t_2) \supset S(t_1) + L(t_1) \leq S(t_2)),\\
&\forall t_1 \forall t_2\ (D(t_1, t_2) \supset S(t_1) + L(t_1) \leq S(t_2) \vee S(t_2) + L(t_2) \leq S(t_1)).
\end{aligned} \tag{4.2}$$

However, one can notice that the specification above is not separated into the theory part

and the propositional part (as required by DPLL(T)). This can be done as follows:

$$\text{“Propositional” part } \phi \text{ is:} \quad \begin{cases} \forall t\ (after(t)), \\ \forall t\ (before(t)), \\ \forall t_1 \forall t_2\ (P(t_1, t_2) \supset prec(t_1, t_2)), \\ \forall t_1 \forall t_2\ (D(t_1, t_2) \supset prec(t_1, t_2) \vee prec(t_2, t_1)). \end{cases} \tag{4.3}$$

$$\text{Theory part } \psi \text{ is:} \quad \begin{cases} after(t) \iff S(t) \geq EST(t), \\ before(t) \iff S(t) + L(t) \leq LET(t), \\ prec(t_1, t_2) \iff S(t_1) + L(t_1) \leq S(t_2). \end{cases}$$

In the real world, the DPLL(T) system works on the propositional level and the program above is first grounded before being fed to the DPLL(T) system. The first part is called “propositional” because the formulas will be turned into propositional ones. In this example we assume that the instance structure $\mathcal{A}$ has domain $A = \{1, 2\}$ and the following interpretations: $EST^{\mathcal{A}} = \{(1:2), (2:2)\}$, $LET^{\mathcal{A}} = \{(1:4), (2:4)\}$, $L^{\mathcal{A}} = \{(1:2), (2:1)\}$, $P^{\mathcal{A}} = \emptyset$, and $D^{\mathcal{A}} = \{(1, 2)\}$. The ground DPLL(T) program for instance structure $\mathcal{A}$ is the

conjunction of ϕ and ψ, where:

$$\text{Propositional part } \phi \text{ is: } \begin{cases} a_1 \\ a_2 \\ b_1 \\ b_2 \\ p_{12} \vee p_{21} \end{cases}$$

$$\text{Theory part } \psi \text{ is: } \begin{cases} a_1 \iff s_1 \geq 1 \\ a_2 \iff s_2 \geq 2 \\ b_1 \iff s_1 \leq 2 \\ b_2 \iff s_2 \leq 3 \\ p_{12} \iff s_1 + 2 \leq s_2 \\ p_{21} \iff s_2 + 1 \leq s_1 \end{cases} \tag{4.4}$$

The DPLL(T) system solves ground program 4.4 as follows: Starting from the empty assignment, the assignment is gradually extended for the set of boolean atoms and, meanwhile, queries to the Theory Solver are made to check whether the current assignment is T-consistent. If not, the theory solver returns a set of literals which are true in the current assignment, but cannot be true together according to theory T and specification ψ. For example consider the partial assignment below:

$$a_1 = a_2 = b_1 = b_2 = p_{21} = \top, p_{12} =? \text{ (unknown)} \tag{4.5}$$

When this set of assignments is passed to the theory solver for difference logic, it can detect that if both a_2 ($s_2 \geq 2$) and b_1 ($s_1 \leq 2$) are true, then $s_2 \geq s_1$. Thus, p_{21} ($s_2+1 \leq s_1$) cannot be true. So, the assignment (4.5) conflicts with the ψ part of ground program (4.4). The reason for this conflict can be described using the set of literals $\{a_2, b_1, p_{21}\}$ saying that they cannot all be true at the same time. Also, the theory solver may even assist the

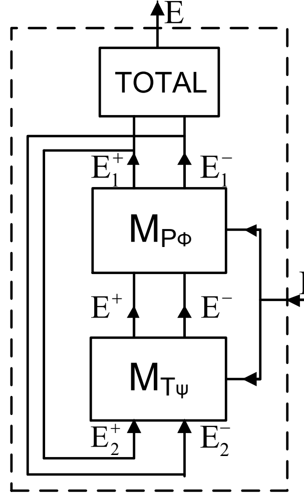

Figure 4.2: Modular System $DPLL(T)_{\phi\wedge\psi}$ Representing the DPLL(T) System on Input Formula $\phi \wedge \psi$

propositional solver by asserting $\neg p_{21}$ *before it is assigned true. For example, once the propositional solver has decided* a_2 *and* b_1 *to be true and not yet made* p_{21} *true, the theory solver can use the fact* $a_2 \wedge b_1 \supset \neg p_{21}$ *(which is a logical* T*-consequence of* ψ*) to assert that* p_{21} *should be false. These two behaviors are modeled in our system through reasons and advice, respectively.*

Next, we show our modular representation of the general DPLL(T) system, and show how the Algorithm 4 on this representation models the solving procedure of the DPLL(T) system. The modular system representing the DPLL(T) system on the input formula $\phi \wedge \psi$ is shown in figure 4.2, where $\sigma = I$, $\varepsilon = E$, and $E^+ \cup E^- \cup E_1^+ \cup E_1^- \cup E_2^+ \cup E_2^-$ is the internal vocabulary of the module. Also, there is feedback from E_1^+ to E_2^+ and from E_1^- to E_2^-. The set of symbols in E^+ and E^- (similarly for E_1^+ and E_1^-, E_2^+ and E_2^-) semantically represents a partial interpretation of the symbols in the expansion vocabulary, i.e., E^+ (resp. E^-) represents the positive (resp. negative) part of the partial interpretation.

Example 16 (Disjoint Scheduling Continued) *Continuing our running example, the*

assignment (4.5) is equivalent to the following partial structure $\mathcal{B}$:

$$\begin{array}{ll} after^{+^{\mathcal{B}}} = \{1,2\} & after^{-^{\mathcal{B}}} = \emptyset \\ before^{+^{\mathcal{B}}} = \{1,2\} & before^{-^{\mathcal{B}}} = \emptyset \\ prec^{+^{\mathcal{B}}} = \{(1,2)\} & prec^{-^{\mathcal{B}}} = \emptyset \end{array}$$

This means that, in our representation of the DPLL(T) modular system, we should have:

$$\{after(1), after(2), before(1), before(2), prec(1,2)\} \subseteq E^{+^{\mathcal{B}}}.$$

There are three MX modules in $DPLL(T)_{\phi\wedge\psi}$. The modules M_{P_ϕ} and M_{T_ψ} work on different parts of the specification. The formula ϕ in M_{P_ϕ} is a CNF representation of the problem specification with all non-propositional literals replaced by new propositional atoms, and the formula ψ in M_{T_ψ} is the formula $\bigwedge_i d_i \Leftrightarrow l_i$ where l_i and d_i are, respectively, an atomic formula in theory T and its associated propositional literal used in M_{P_ϕ}. The module M_{P_ϕ} is the set of structures $\mathcal{B}$ such that:

$$(E_1^{+^{\mathcal{B}}}, E_1^{-^{\mathcal{B}}}) = \begin{cases} (D, D) & \text{if } R^+ \cap R^- \neq \emptyset \\ (R^+, R^{+^c}) & \text{if } R^+ \cap R^- = \emptyset, I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup R^+ \cup \neg R^- \models \phi \\ (R^+, R^-) & \text{if } R^+ \cap R^- = \emptyset, I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup R^+ \cup \neg R^- \not\models \phi \end{cases},$$

where $D = B^n$, n is the arity of E^+, and (R^+, R^-) is the result of Unit Propagation on ϕ under $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup E^{+^{\mathcal{B}}} \cup \neg E^{-^{\mathcal{B}}}$.

Example 17 (Disjoint Scheduling Continued) *Continuing our running example, assuming that $E^{+^{\mathcal{B}}} = E^{-^{\mathcal{B}}} = \emptyset$, in order for the module M_{P_ϕ} to accept the structure $\mathcal{B}$, we should have that $E_1^{+^{\mathcal{B}}} = \{after(1), after(2), before(1), before(2)\}$. This is because R^+ (the positive atoms deduced by unit propagation on ϕ) asserts that a_1, a_2, b_1 and b_2 should all be true (look at the propositional part of Equation (4.4)).*

Similarly, the module M_{T_ψ} is defined as the set of structures $\mathcal{B}$ such that:

$$(E^{+\mathcal{B}}, E^{-\mathcal{B}}) = \begin{cases} (D, D) & \text{if } R^+ \cap R^- \neq \emptyset \\ (D, D) & \text{if } R^+ \cap R^- = \emptyset, I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup R^+ \cup \neg R^- \models_T \neg\psi \\ (R^+, R^{+^c}) & \text{if } R^+ \cap R^- = \emptyset, I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup R^+ \cup \neg R^- \models_T \psi \\ (R^+, R^-) & \text{if } R^+ \cap R^- = \emptyset, \text{T-satisfiability unknown} \end{cases},$$

where D is as before and (R^+, R^-) is the result of Theory Propagation on ψ under $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup E_2^{+\mathcal{B}} \cup \neg E_2^{-\mathcal{B}}$, and $R \models_T \psi$ denotes that ψ is T-satisfiable under the set of facts R. Note that the satisfiability test is not necessarily complete. It can be done in different degrees depending on the complexity of different theories, e.g., exhaustive theory propagation could be applied for low complexity theories like Theory of Difference Logic, and non-exhaustive theory propagation for more complex theories like Theory of Equality with Uninterpreted Functions (EUF) [45].

Example 18 (Disjoint Scheduling Continued) *Continuing our running example, assume that* $E_2^{+\mathcal{B}} = \{after(1), after(2), before(1), before(2)\}$ *and* $E_2^{-\mathcal{B}} = \emptyset$*. In order for the module* M_{T_ψ} *to accept the structure* $\mathcal{B}$*, we should have that* $E^{-\mathcal{B}} = \{prec(1, 2)\}$*. This is because* R^- *(the negative facts deduced by* T*-propagation on* ψ*) tells us that if* a_2 *and* b_1 *are true (which they are), then* p_{12} *should be false.*

The module $TOTAL$ is the set of structures $\mathcal{B}$ such that $E_1^{+\mathcal{B}} \cap E_1^{-\mathcal{B}} = \emptyset$, $E_1^{+\mathcal{B}} \cup E_1^{-\mathcal{B}} = D$, and $E_1^{+\mathcal{B}} = E^{\mathcal{B}}$.

We define the modular system $DPLL(T)_{\phi\wedge\psi}$ as:

$$DPLL(T)_{\phi\wedge\psi} := \pi_{\{I,E\}}(((M_{T_\psi} \rhd M_{P_\phi})[E_1^+ = E_2^+][E_1^- = E_2^-]) \rhd TOTAL). \tag{4.6}$$

To show that the combined module $DPLL(T)_{\phi\wedge\psi}$ is correct, we prove that a structure is in the modular system $DPLL(T)_{\phi\wedge\psi}$ iff it satisfies both formula ϕ and ψ. Consider any model of the modular system. Note that for both modules M_{P_ϕ} and M_{T_ψ}, the outputs always contain all the information that the inputs have, i.e., for any structure $\mathcal{B}$ in the module M_{P_ϕ}, we have $E_1^{+\mathcal{B}} \supseteq E^{+\mathcal{B}}$ and $E_1^{-\mathcal{B}} \supseteq E^{-\mathcal{B}}$, and for any structure $\mathcal{B}$ in M_{T_ψ}, we have $E^{+\mathcal{B}} \supseteq E_2^{+\mathcal{B}}$ and $E^{-\mathcal{B}} \supseteq E_2^{-\mathcal{B}}$. Furthermore, from the semantics of the feedback operator, we know that $E_1^{+\mathcal{B}} = E_2^{+\mathcal{B}}$ and $E_1^{-\mathcal{B}} = E_2^{-\mathcal{B}}$. Thus, we have $E^{+\mathcal{B}} = E_1^{+\mathcal{B}} = E_2^{+\mathcal{B}}$

and $E^{-\mathcal{B}} = E_1^{-\mathcal{B}} = E_2^{-\mathcal{B}}$. Moreover, from the definition of module $TOTAL$, we know that $(E_1^{+\mathcal{B}}, E_1^{-\mathcal{B}})$ represents a total interpretation of the symbols in E and $E^{\mathcal{B}} = E_1^{+\mathcal{B}}$. Finally, from the definitions of M_{P_ϕ} and M_{T_ψ} on encodings of total interpretations, we can conclude that $\mathcal{B} \models \phi$ and $\mathcal{B} \models_T \psi$. On the other hand, it is easy to see that for any structure $\mathcal{B}$ such that $\mathcal{B} \models \phi$ and $\mathcal{B} \models_T \psi$, $\mathcal{B}$ is in $DPLL(T)_{\phi\wedge\psi}$.

So, there is a one-to-one correspondence between models of $DPLL(T)_{\phi\wedge\psi}$ and the propositional part of the solutions to the DPLL(T) system on input formula $\phi \wedge \psi$. To find a solution, one can compute a model of this modular system.

To solve $DPLL(T)_{\phi\wedge\psi}$, we introduce a solver S to be any DPLL-based online SAT solver, so that it performs the basic actions of Decide, UnitPropagate, Fail, Restart, and also Backjump when the backjumping clause is added to the solver. The three modules $TOTAL$, M_{T_ψ} and M_{P_ϕ} are attached with oracles O_{TOTAL}, O_T and O_P respectively. They accept a partial structure $\mathcal{B}$ iff their respective module constraints are not falsified by $\mathcal{B}$.

Example 19 (Disjoint Scheduling Continued (O_P, O_T and O_{TOTAL})) *Let ϕ and ψ in Figure 4.2 be, respectively, the propositional part and theory part of the specification in Example 15. Let the structure $\mathcal{B}$ contain the same set of partial assignments as the one in Example 15, i.e., $after_1^{+\mathcal{B}} = before_1^{+\mathcal{B}} = \{1,2\}$, $prec_1^{+\mathcal{B}} = \{(2,1)\}$, and $after_1^{-\mathcal{B}} = before_1^{-\mathcal{B}} = prec_1^{-} = \emptyset$. When O_T is queried on $\mathcal{B}$, it returns $after_1^{+}(2) \wedge before_1^{+}(1) \supset prec^{-}(2,1)$ as the advice to the solver S. Together with the advice $prec^{-}(2,1) \supset prec_1^{-}(2,1)$ from the oracle O_P, in the next round, S will conclude $prec_1^{-}(2,1)$ to be true. This new structure from S will be rejected by the oracle O_{TOTAL} with the reason $prec_1^{-}(2,1) \supset \neg prec_1^{+}(2,1)$.*

Detailed constructions for the solver S, oracle O_{TOTAL}, oracle O_T and O_P follows:

Solver S is a DPLL-based SAT solver (clearly complete and online).

Oracle O_{TOTAL} accepts a partial structure $\mathcal{B}$ iff $E_1^{+\mathcal{B}} \cap E_1^{-\mathcal{B}} = \emptyset$, $E_1^{+\mathcal{B}} \cup E_1^{-\mathcal{B}} = D$, and $E^{\mathcal{B}} = E^{+\mathcal{B}}$. If $\mathcal{B}$ is rejected, O_{TOTAL} returns $\bigwedge_{\omega\in\Omega'} \omega$ as the reason, where Ω' is any non-empty subset of the set $\Omega = \{E_1^{+}(d) \Leftrightarrow \neg E_1^{-}(d) \mid d \in D, \mathcal{B} \not\models E_1^{+}(d) \Leftrightarrow \neg E_1^{-}(d)\} \cup \{E(d) \Leftrightarrow E_1^{+}(d) \mid d \in D, \mathcal{B} \not\models E(d) \Leftrightarrow E_1^{+}(d)\}$. O_{TOTAL} returns the set Ω as the set of advices when $\mathcal{B}$ is the empty expansion of the instance structure, and the empty set otherwise.[2] Clearly, O_{TOTAL} is a CCAV oracle.

Oracle O_T accepts a partial structure $\mathcal{B}$ iff it does not falsify the constraints described

[2]This makes sure that Ω is returned only once at the beginning.

above for module M_{T_ψ} on I, E^+, E^-, E_2^+, and E_2^-. Let (R^+, R^-) denote the result of the Theory Propagation on ψ under $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup {E_2^+}^{\mathcal{B}} \cup \neg {E_2^-}^{\mathcal{B}}$. Then, if $\mathcal{B}$ is rejected,

1. If $R^+ \cap R^- \neq \emptyset$ or ψ is T-unsatisfiable under $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup R^+ \cup \neg R^-$, O_T returns a reason ω of the form $\bigwedge_{d \in D_1} E_2^+(d) \wedge \bigwedge_{d \in D_2} E_2^-(d) \supset \bigwedge_{d \in D_3}(E^+(d) \wedge E^-(d))$ with $D_1 \subseteq D$, $D_2 \subseteq D$, $\emptyset \subsetneq D_3 \subseteq D$, $T \models \bigvee_{d \in D_1} \neg l(d) \vee \bigvee_{d \in D_2} l(d)$, and $\mathcal{B} \models \neg\omega$, where $l(d)$ denotes the atomic formula l in ψ whose associated propositional atom is d. Note that from the advices and reasons from oracles, the solver can understand that right hand side of the implication is inconsistent, and thus the reason corresponds to the set of T-inconsistent literals from the theory solver in the DPLL(T) system.

2. Else if ψ is T-satisfiable under $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup R^+ \cup \neg R^-$, O_T returns a reason ω of the form $\bigwedge_{d \in D_1} E_2^+(d) \wedge \bigwedge_{d \in D_2} E_2^-(d) \supset \bigwedge_{d \in R^+} E^+(d) \wedge \bigwedge_{d \in {R^+}^c} E^-(d)$, where $D_1 \subseteq D$, $D_2 \subseteq D$, and $\mathcal{B} \models \neg\omega$.

3. Else, O_T returns a reason similar to the second case except that it uses R^- instead of ${R^+}^c$.

By the definition of M_{T_ψ}, we know that $\mathcal{B}$ falsifies the reason and all models of M_{T_ψ} satisfy the reason. Thus, O_T is complete and constructive. O_T may also return some advices in the same form as any ω above such that $\mathcal{B}$ satisfies the left hand side of the implication, but not the right hand side. Also, since the outputs of M_{T_ψ} always subsume the inputs, O_T may also return the set $\{E_2^+(d) \supset E^+(d) \mid d \in D,\ \mathcal{B} \models E_2^+(d), \mathcal{B} \not\models E^+(d)\} \cup \{E_2^-(d) \supset E^-(d) \mid d \in D, \mathcal{B} \models E_2^-(d), \mathcal{B} \not\models E^-(d)\}$ as the set of advices.[3] Clearly, all the structures in M_{T_ψ} satisfy all sets of advices. Hence, O_T is an advising oracle. Finally, O_T always makes the correct decision for a total structure and rejects a partial structure only when it falsifies the constraints for M_{T_ψ}. Oracle O_T never rejects any good partial structure $\mathcal{B}$ (although it may accept some bad non-total structures). Therefore, O_T is a verifying oracle.

Oracle O_P accepts a partial structure $\mathcal{B}$ iff it does not falsify the constraints for module M_{P_ϕ} on I, E^+, E^-, E_1^+, and E_1^-. Let (R^+, R^-) denote the result of the Unit Propagation on ϕ under $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup {E^+}^{\mathcal{B}} \cup \neg {E^-}^{\mathcal{B}}$. Then, if $\mathcal{B}$ is rejected,

1. If $R^+ \cap R^- \neq \emptyset$, O_P returns a reason ω of the form $\bigwedge_{d \in D_1} E^+(d) \wedge \bigwedge_{d \in D_2} E^-(d) \supset \bigwedge_{d \in D_3}(E_1^+(d) \wedge E_1^-(d))$ with $D_1 \subseteq D$, $D_2 \subseteq D$, $\emptyset \subsetneq D_3 \subseteq D$, $\phi \models \bigvee_{d \in D_1} \neg d \vee \bigvee_{d \in D_2} d$

[3] Again O_T only returns this set when $\mathcal{B}$ is the empty expansion of the instance structure.

and $\mathcal{B} \models \neg\omega$.

2. Else if $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup R^+ \cup \neg R^- \models \phi$, O_P returns a reason ω of the form $\bigwedge_{d\in D_1} E^+(d) \wedge \bigwedge_{d\in D_2} E^-(d) \supset \bigwedge_{d\in R^+} E_1^+(d) \wedge \bigwedge_{d\in {R^+}^c} E_1^-(d)$, where $D_1 \subseteq D$, $D_2 \subseteq D$, and $\mathcal{B} \models \neg\omega$.

3. Else, O_P returns a reason similar to the second case except that it uses R^- instead of ${R^+}^c$.

O_P may return the set of advices in the same form as any ω above such that $\mathcal{B}$ satisfies the left hand side of the implication, but not the right hand side. Also, since the outputs of M_{P_ϕ} always subsume the inputs, O_P may also return the set $\{E^+(d) \supset E_1^+(d) \mid d \in D, \mathcal{B} \models E^+(d), \mathcal{B} \not\models E_1^+(d)\} \cup \{E^-(d) \supset E_1^-(d) \mid d \in D, \mathcal{B} \models E^-(d), \mathcal{B} \not\models E_1^-(d)\}$ as the set of advices.

Proposition 3 *1. Modular system $DPLL(T)_{\phi\wedge\psi}$ is the set of structures $\mathcal{B}$ such that $\mathcal{B} \models \phi$ and $\mathcal{B} \models_T \psi$.*

2. Solver S is complete and online.

3. O_P, O_T, and O_{TOTAL} are CCAV oracles.

4. Algorithm 4 on modular system $DPLL(T)_{\phi\wedge\psi}$ associated with oracles O_P, O_T, O_{TOTAL}, and the solver S models the solving procedure of the DPLL(T) system on input formula $\phi \wedge \psi$.

The DPLL(T) architecture is known to be very efficient and many solvers use it, including most SMT solvers [51]. The DPLL(Agg) module [12] is suitable for all DPLL-based SAT, SMT and ASP solvers to check satisfiability of aggregate expressions in DPLL(T) contexts. All these systems are representable in our modular framework.

4.4.2 Modelling ILP Solvers

Integer Linear Programming solvers solve optimization problems. In this thesis, we model ILP solvers which use general branch-and-cut method to solve *search* problems instead, i.e., when the target function is a constant. Throughout the section, we use the following MX task as a running example to show how Algorithm 4 models branch-and-cut based ILP solvers.

Example 20 (Facility Opening Problem) *Given a set of facilities $F = \{1, 2, \cdots, n\}$, a set of clients $C = \{1, 2, \cdots, m\}$, a function $E(f, c)$ denoting whether the facility f is available to the client c, a function $OC(f)$ indicating the cost of opening the facility f, a function $UC(c, f)$ representing the cost of the client c using the facility f, and a constant d, the task is to open a subset of the facilities ($O(f)$) and assign open facilities to available clients ($U(c, f)$) such that each client has at least one open facility assigned, and the total cost (both opening cost and using cost) does not exceed d. Above, we have $\sigma = \{available, OC, UC, d\}$, and $\varepsilon = \{O, U\}$.*

Example 20 describes a famous problem of facility opening where you want to minimize the total cost of opening and using facilities so that all your clients are covered by at least one facility. This problem showcases some of the strengths of ILP solvers. First, we describe how ILP solvers utilize the general branch-and-cut algorithm [47] to tackle such problems. The general algorithm is:

1. Initialization: $S = \{\text{ILP}^0\}$ with ILP^0 the initial problem.
2. Termination: If $S = \emptyset$, return UNSAT.
3. Problem Select: Select and remove problem ILP^i from S.
4. Relaxation: Solve LP relaxation of ILP^i (as a search problem). If infeasible, go to step 2. Otherwise, if solution X^{iR} of LP relaxation is integral, return solution X^{iR}.
5. Add Cutting Planes: Add a cutting plane violating X^{iR} to relaxation and go to 4.
6. Partitioning: Find partition $\{C^{ij}\}_{j=1}^{j=k}$ of constraint set C^i of problem ILP^i. Create k subproblems ILP^{ij} for $j = 1, \cdots, k$, by restricting the feasible region of subproblem ILP^{ij} to C^{ij}. Add those k problems to S and go to step 2. Often, in practice, finding a partition is simplified by picking a variable x_i with non-integral value v_i in X^{iR} and returning partition $\{C^i \cup \{x_i \leq \lfloor v_i \rfloor\}, C^i \cup \{x_i \geq \lceil v_i \rceil\}\}$.

In order for the branch-and-cut algorithm above to solve the problem in Example 20, we must describe the example using a set of linear inequalities as follows.

Example 21 (Facility Opening (Specification, Instance, Ground Program)) *We use the following high level ILP specification to represent the facility opening problem*

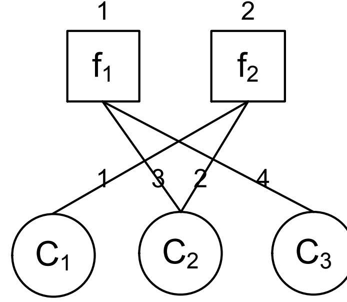

Figure 4.3: Facility Opening Problem Instance

in Example 20.

$$
\begin{aligned}
&\textstyle\sum_f OC(f) * O(f) + \sum_{c,f} UC(c,f) * U(c,f) \leq d \\
&\forall c \forall f \ U(c,f) \leq O(f) \\
&\forall c \forall f \ U(c,f) \leq E(f,c) \\
&\forall c \ \textstyle\sum_f uses(c,f) \geq 1 \\
&\forall c \forall f \ 0 \leq U(c,f) \leq 1 \\
&\forall f \ 0 \leq O(f) \leq 1
\end{aligned} \tag{4.7}
$$

However, the specification in Equation 4.7 contains a set of quantifiers which must first be expanded before giving the program into an ILP solver. Next we show how such a task can be accomplished given a problem instance.

A problem instance is shown in Figure 21. In Figure 21, two facilities are shown on the top and three clients are shown below. The availability of facilities to clients is represented as edges between facilities and clients. The opening costs for facilities are shown on top of facilities, and the costs of use are shown on the edges. The ground ILP program for this instance is as follows:

$$U_{1,1} \leq 0 \qquad U_{1,2} \leq 1$$

$$U_{2,1} \leq 1 \qquad U_{2,2} \leq 1$$

$$U_{3,1} \leq 1 \qquad U_{3,2} \leq 0$$

$$U_{1,1} + U_{1,2} \geq 1 \qquad U_{2,1} + U_{2,2} \geq 1 \qquad U_{3,1} + U_{3,2} \geq 1$$

$$U_{1,1} - O_1 \leq 0 \qquad U_{1,2} - O_2 \leq 0$$

$$U_{2,1} - O_1 \leq 0 \qquad U_{2,2} - O_2 \leq 0$$

$$U_{3,1} - O_1 \leq 0 \qquad U_{3,2} - O_2 \leq 0$$

$$0.5\ O_1 + 1.5\ O_2 + U_{1,2} + 3\ U_{2,1} + 2\ U_{2,2} + 4\ U_{3,1} \leq 10$$

$$0 \leq U_{1,1}, U_{1,2}, U_{2,1}, U_{2,2}, U_{3,1}, U_{3,2}, O_1, O_2 \leq 1$$

Now, consider one of the non-integral solutions of this as follows:

$$U_{1,1} = 0, U_{1,2} = 1, U_{2,1} = \frac{1}{3}, U_{2,2} = 1, U_{3,1} = 1, U_{3,2} = 0, O_1 = 1, O_2 = 1$$

From the description of a branch-and-cut ILP solver above, this solution can be discarded either by performing partitioning or adding a cutting plane to the set of linear constraints.

Partitioning: $U_{2,1} \leq 0 \vee U_{2,1} \geq 1$

Cutting plane: $U_{2,1} + U_{2,2} \leq 1$

Note that the non-integral solution above does not satisfy any of the two conditions while all integral solutions satisfy both of them.

Next, we describe how we construct the modular system representing the ILP solver, and show how our algorithm on the modular system models the solving procedure of the ILP solver. We use the modular system shown in Figure 4.4 to represent the ILP solver solving the problem axiomatized in the specification ϕ. The specification is shared among the modules and also among the oracles associated to the modules. The module C_ϕ takes a set of variable assignments F_1 and a set of cutting planes SC_1 as inputs and returns another set of cutting planes SC_2. When all the assignments in F_1 are integral, SC_2 is equal to

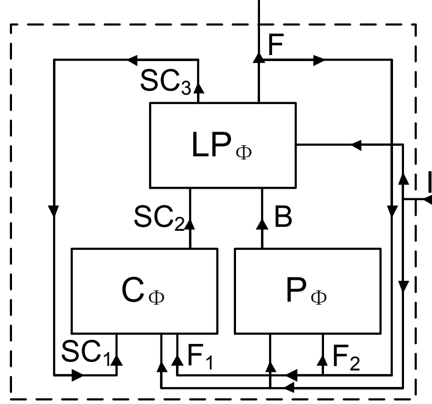

Figure 4.4: Modular System Representing an ILP Solver

SC_1, and if not, SC_2 is the union of SC_1 and a cutting plane violated by F_1 w.r.t. the set of linear constraints $SC_1 \cup \phi$. The module P_ϕ takes a set of assignments F_2 as input and outputs a set of range constraints $B = \{B_x \mid F_2(x) \notin \mathcal{Z}\}$, where B_x is non-deterministically chosen from the set $\{x \leq \lfloor F_2(x) \rfloor,\ x \geq \lceil F_2(x) \rceil\}$. The module LP_ϕ takes the set of cutting planes SC_2 and the set of range constraints B as inputs and outputs the set of cuttings planes SC_3 and the set of assignments F in a deterministic way such that SC_3 is the union of SC_2 and B, and F is a total assignment satisfying $SC_2 \cup B \cup \phi$. LP_ϕ is undefined when $SC_2 \cup B \cup \phi$ is inconsistent. We define the compound module ILP_ϕ to be:

$$ILP_\phi := \pi_{\{F\}}(((C_\phi \cap P_\phi) \rhd LP_\phi)[SC_3 = SC_1][F = F_1][F = F_2]).$$

To show that the combined module ILP_ϕ is correct, consider any model of the modular system. By the definition of LP_ϕ, we know that F satisfies ϕ. Furthermore, the set B is empty in the model because F satisfies all the linear constraints in B, but F_2 (which is equal to F by the semantics of feedback operator) falsifies those constraints. Thus by the definition of the module P_ϕ, we know that F_2 (also F) is integral. Thus F is an integral solution to ϕ. On the other hand, for any integral solution S to ϕ, consider a structure $\mathcal{B}$ such that $F^{\mathcal{B}} = F_1^{\mathcal{B}} = F_2^{\mathcal{B}} = S$, $B^{\mathcal{B}} = \emptyset$, and $SC_1^{\mathcal{B}} = SC_2^{\mathcal{B}} = SC_3^{\mathcal{B}} = \bigcup_x \{x \leq F(x),\ x \geq F(x)\}$. Then clearly, $\mathcal{B}$ is in the module ILP_ϕ, i.e., $\mathcal{B}$ is the model of the module ILP_ϕ.

So there is one-to-one correspondence between the solutions of the ILP problem with input ϕ, and the models of the modular system ILP_ϕ. We compute a model of this modular system by associating modules with oracles (O_c, O_p and O_{lp}) and introducing a solver S that

interacts with those oracles. Each oracle rejects a partial structure $\mathcal{B}$ if it contradicts the corresponding module definition and in this case, the reason for the rejection is provided.

Example 22 (Facility Opening Problem Continued (O_p and O_c))
Let ϕ in Figure 4.4 be the specification shown in Example 21. Let $F^{\mathcal{B}}$ contain the same non-integral solution as the one in Example 21, i.e., $F^{\mathcal{B}} = F_1^{\mathcal{B}} = F_2^{\mathcal{B}} = \{U(1,1) = 0, U(1,2) = 1, U(2,1) = 1/3, U(2,2) = 1, U(3,1) = 0, U(3,2) = 0, O(1) = 1, O(2) = 1\}$, and let $B^{\mathcal{B}} = {SC_1}^{\mathcal{B}} = {SC_2}^{\mathcal{B}} = {SC_3}^{\mathcal{B}} = \emptyset$. As shown in Example 21, this non-integral solution can be eliminated by either partitioning or adding a cutting plane violating the set of assignments. The partitioning is modelled by O_p rejecting the structure $\mathcal{B}$ and returning the reason $B("U(2,1) \leq 0") \vee B("U(2,1) \geq 1")$ [4] ; and the cutting plane method can be modelled by O_c rejecting $\mathcal{B}$ with the reason $SC_2("U(2,1) + U(2,2) \leq 1")$.

The LP solving in ILP solver is modelled by the oracle O_{lp} for the module LP_ϕ.

Example 23 (Facility Opening Problem Continued (O_{lp})) *Let $F^{\mathcal{B}}$, $F_1^{\mathcal{B}}$, and $F_2^{\mathcal{B}}$ contain the same non-integral solution as in Example 22, but let $B^{\mathcal{B}} = \{"U(2,1) \leq 0"\}$ and $SC_2 = \{"U(2,1) + U(2,2) \leq 1"\}$. Note that the non-integral solution violates both constraints $U(2,1) \leq 0$ and $U(2,1) + U(2,2) \leq 1$. Then O_{lp} rejects $\mathcal{B}$ with the reason either being $B("U(2,1) \leq 0") \supset F("U(2,1)") \leq 0$ (for violating the partitioning constraint), or being $SC_2("U(2,1) + U(2,2) \leq 1") \wedge F_1("U(2,2)") > 1 \supset F_1("U(2,1)") < 0$ (for violating the cutting plane). This way, O_{lp} guides the assignments F to satisfy all the constraints in ${SC_2}^{\mathcal{B}}$ and $B^{\mathcal{B}}$.*

Next, we give the formal constructions of the solver and the oracles.

Solver S accepts the full propositional language with atomic formulas being either boolean variables or range constraints. In addition, S can assign numerical values (for F), according to the set of derived range constraints.

Oracle O_p accepts a partial structure $\mathcal{B}$ if it does not falsity the constraints described above for module P_ϕ on F_2 and B. If $\mathcal{B}$ is rejected and ${F_2}^{\mathcal{B}}$ is non-integral, O_p returns the reason $B("F_2(x) \leq \lfloor v \rfloor") \vee B("F_2(x) \geq \lceil v \rceil")$, where v is equal to ${F_2}^{\mathcal{B}}(x)$ and is non-integral.

Oracle O_c accepts a partial structure $\mathcal{B}$ if it does not falsify the constraints described above on F_1, SC_1, and SC_2 for the C_ϕ module. If $\mathcal{B}$ is rejected, O_c returns the reason

[4] *As the specification ϕ is shared between the module and the oracle, O_p can also return $B("U(2,1) = 0") \vee B("U(2,1) = 1")$ as the reason.*

$\bigwedge_i F_1(x_i) = v_i \supset \bigwedge_{c \in SC_1 \Delta SC_2}(SC_1(c) \iff SC_2(c))$ when $F_1^{\mathcal{B}}$ is integral, and the reason $(\bigwedge_{c \in I} SC_1(c)) \wedge (\bigwedge_i F_1(x_i) = v_i) \supset SC_2(c')$, where $I \subseteq SC_1 \cup \phi$, F_1 is the only intersection of the set of linear constraints I, and c' is the cutting plane on I that violates F_1.

Oracle O_{lp} accepts a partial structure $\mathcal{B}$ if it does not falsify the constraints of the module LP_ϕ on SC_2, B, SC_3, and F. If $\mathcal{B}$ is rejected, O_{lp} returns the reason of the form $\psi = (\bigwedge_{c \in SC_2{}^{\mathcal{B}}} SC_2(c)) \wedge (\bigwedge_{c \in B^{\mathcal{B}}} B(c)) \supset (\bigwedge_{c \in SC'_3} SC_3(c)) \wedge (\bigwedge_x F(x))$, such that $SC'_3 \subseteq SC_2{}^{\mathcal{B}} \cup B^{\mathcal{B}}$, the new assignments to F satisfy $SC_2{}^{\mathcal{B}} \cup B^{\mathcal{B}} \cup \phi$, and $\mathcal{B} \not\models \psi$.

No advices are needed from any oracles in order to model the branch-and-cut ILP solvers. Thus, all oracles always return the empty set as the set of advices.

Proposition 4

1. *Modular system ILP_ϕ is the set of structures representing the sets of integral solutions of ϕ.*
2. *S is complete and online.*
3. *O_c, O_p and O_{lp} are CCAV oracles.*
4. *Algorithm 4 on modular system ILP_ϕ, associated with oracles O_c, O_p, O_{lp}, and the solver S models the branch-and-cut-based ILP solver on the input formula ϕ.*

There are many other solvers in the ILP community that use some ILP or MILP solver as their low-level solver. It is not hard to observe that most of them also have similar architectures that can be closely mapped to our algorithm.

4.4.3 Modelling Constraint Answer Set Solvers

The Answer Set Programming (ASP) community puts a lot of effort into optimizing their solvers. One such effort addresses ASP programs with variables ranging over huge domains (for which, ASP solvers alone perform poorly due to the huge memory needed). However, embedding Constraint Programming (CP) techniques into ASP solving is proved useful because complete grounding can be avoided.

In [9], the authors extend the language of ASP and its reasoning method to avoid grounding of variables with large domains by using constraint solving techniques. The algorithm uses ASP and CP solvers as black boxes and non-deterministically extends a partial solution to the ASP part and checks it with the CP solver. Also, in [40], the

authors integrate answer set generation and constraint solving using a traditional DPLL-like backtracking algorithm which embeds a CP solver into an ASP solver.

Recently, the authors of [30] developed an improved hybrid Constraint Answer Set Programming (CASP) solver which supports advanced backjumping and conflict-driven nogood learning (CDNL) techniques. They show that their solver's performance is comparable to state-of-the-art SMT solvers. In [30], a partial grounding is applied before running the algorithm, thus, the algorithm in [30] is on a propositional level. In addition, instead of directly computing the answer set of the ASP program, the authors compute a boolean assignment satisfying a set of nogoods obtained from the Clark completion of the ASP program and from the loop formulas [29]. This enables them to be able to apply solving technology from the areas of CSP and SAT, e.g., conflict-driven learning, backjumping, watched literals, etc. A brief description of this algorithm follows: Starting from an empty set of assignments and derived nogoods, the algorithm gradually extends the partial assignments by both unit propagation in ASP [31] and constraint propagation in CP [50]. If a conflict occurs (during either unit propagation or constraint propagation), a nogood containing the corresponding unique implication point (UIP) [41, 39] is learned [5] and the algorithm backjumps to the decision level of the UIP. Otherwise, the algorithm decides on the truth value of one of the currently unassigned atoms and continues to apply the propagation. If the assignment becomes total, the CP oracle queries to check whether this is indeed a solution for the corresponding constraint satisfaction problem (CSP). This step is necessary because simply performing constraint propagation on the set of constraints is not sufficient to decide the feasibility of constraints.

In this section, following MX task is used as a running example to illustrate how Algorithm 4 can model above CASP solver. To improve the readability, all examples in this section is axiomatized in ASP program, instead of its completion and loop formulas as in the CASP solver.

Example 24 (Planning with Cumulative Scheduling) *Given a set of tasks,* $Task = \{t_1, \cdots, t_n\}$*, a set of states,* $\{s_1, \cdots, s_m\}$*, a predicate* $CS(t, s_1, s_2)$ *saying that performing task* t *changes the state from* s_1 *to* s_2*, a starting state* S*, and a goal state* G*, the*

[5] Practical CP solvers do not provide reasons for rejecting partial structures. This issue is dealt with in [30] by wrapping CP solvers with a conflict analysis mechanism to compute nogoods based on the first UIP scheme.

goal is to perform a set of tasks to get to the goal state from the starting state. In addition, each task t_i has an earliest starting time $EST(t_i)$, a latest ending time $LET(t_i)$, a duration $D(t_i)$, and the amount of resources it needs ($R(t_i)$). The tasks could be performed simultaneously, but the total amount of resources occupied at any time should not exceed the total available resources TR. Let $do(t_i)$ denote that the task t_i is performed, then $\sigma = \{CS, S, G, EST, LET, D, R, TR\}$ and $\varepsilon = \{do\}$.

We axiomatize the problem in Example 24 in ASP, together with the *cumulative* constraint in Constraint Programming. The *cumulative* constraint may be written

$$cumulative(est, let, d, r, R),$$

where the arguments represent, respectively, the set of earliest starting time of the tasks, the set of latest ending time of the task, the set of duration of the tasks, the set of resource consumption of the tasks, and the amount of available resources. It returns true only when there is a feasible scheduling that respects all the specified constraints.

Example 25 (Planning Continued (Specification, Instance, Ground Program)) *Following CASP specification is used to represent the planning with cumulative scheduling problem* [6].

$$
\begin{aligned}
&0\{do(t)\}1 \leftarrow Task(t).\\
&reaches(s_2) \leftarrow do(t), CS(t, s_1, s_2), reaches(s_1).\\
&reaches(S).\\
&\leftarrow not\ reaches(G).\\
&\leftarrow not\ CP :: cumulative(\{EST(t) : do(t)\},\\
&\qquad\qquad \{LET(t) : do(t)\}, \{D(t) : do(t)\}, \{R(t) : do(t)\}, TR).
\end{aligned}
\tag{4.8}
$$

Consider the instance shown in figure 25 with the set of tasks $\{t_1, t_2, t_3, t_4\}$ and the set of states $\{S, U, V, G\}$. The CS relation is shown as the edges between two states, i.e.,

[6] *This specification does not necessarily follow the syntax of any specific system.*

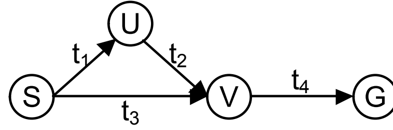

Figure 4.5: Planning with Cumulative Scheduling Problem Instance

$CS^{\mathcal{A}} = \{(t_1, S, U), (t_2, U, V), (t_3, S, V), (t_4, V, G)\}$. *Moreover, let* $EST^{\mathcal{A}} = \{(t_1 : 0), (t_2 : 0), (t_3 : 0), (t_4 : 0)\}$, $LET^{\mathcal{A}} = D^{\mathcal{A}} = \{(t_1 : 2), (t_2 : 2), (t_3 : 2), (t_4 : 2)\}$, $R^{\mathcal{A}} = \{(t_1 : 1), (t_2 : 2), (t_3 : 4), (t_4 : 4)\}$, *and* $TR^{\mathcal{A}} = 7$. *Then the ground program corresponding to this instance is as follows:*

$$\text{ASP part } \phi \text{ is: } \begin{cases} 0\{do(t_1)\}1. \\ 0\{do(t_2)\}1. \\ 0\{do(t_3)\}1. \\ 0\{do(t_4)\}1. \\ reaches(U) \leftarrow do(t_1), reaches(S). \\ reaches(V) \leftarrow do(t_2), reaches(U). \\ reaches(V) \leftarrow do(t_3), reaches(S). \\ reaches(G) \leftarrow do(t_4), reaches(V). \\ reaches(S). \\ \leftarrow not\ reaches(G). \\ \leftarrow not\ C \end{cases}$$

$$\text{CP part } \psi \text{ is: } \begin{cases} C \Leftrightarrow cumulative(\{0 : do(t_1), 0 : do(t_2), 0 : do(t_3), 0 : do(t_4)\}, \\ \qquad \{2 : do(t_1), 2 : do(t_2), 2 : do(t_3), 2 : do(t_4)\}, \\ \qquad \{2 : do(t_1), 2 : do(t_2), 2 : do(t_3), 2 : do(t_4)\}, \\ \qquad \{1 : do(t_1), 2 : do(t_2), 4 : do(t_3), 4 : do(t_4)\}, 7) \end{cases} \tag{4.9}$$

The CASP system solves the ground program 4.9 in a similar way to the DPLL(T) system described in Section 4.4.1: Stating from the empty assignment, the CASP solver gradually computes the answer set and meanwhile, queries to the CP solver to check whether the set of constraint corresponding to the current assignment is consistent. If not, the CP solver returns a set of literals that cannot be true together. For example, consider the partial assignment below:

$$do(t_3) = do(t_4) = reaches(S) = reaches(V) = C = \top, the\ others\ unknown$$

When the CP solver gets this set of assignments, it can deduce that the cumulative constraint (C) cannot be true based on the assignments, because all the tasks have the same earliest starting time and the latest ending time with the intervals the same as the durations, which enforces all the tasks being scheduled at the same time. However, t_3 and t_4 cannot be scheduled simultaneously as they together require 8 resources while only 7 resources are available. The reason for this conflict can be described using the set of literals $\{C, do(t_3), do(t_4)\}$. On the other hand, before the ASP solver decides to schedule both t_3 and t_4, the CP solver may return the fact $C \wedge do(t_3) \supset \neg do(t_4)$ to prevent the two tasks from both being performed. These two behaviors are modelled later in the section through reasons and advices, respectively.

Next, we show our modular representation of the CASP solver and illustrate how the Algorithm 4 on this representation models the solving procedure of the CASP system. The modular system we use to represent the CASP solver is very similar to the one in Figure 4.2 (for the DPLL(T) system), except that we have module ASP_ϕ instead of M_{P_ϕ} and CP_ψ instead of M_{T_ψ}.

Similar to the modules M_{P_ϕ} and M_{T_ψ} in Figure 4.2, ASP_ϕ and CP_ψ work on different parts of the specification. The formula ϕ in ASP_ϕ corresponds to the ASP program with all CP constraints replaced by propositional literals, and the formula ψ in CP_ψ is the formula $\bigwedge_i d_i \Leftrightarrow l_i$ where l_i and d_i are, respectively, an atomic CP constraint and its associated propositional atom used in ASP_ϕ.

The module ASP_ϕ is the set of structures $\mathcal{B}$ such that:

$$(E_1^{+\mathcal{B}}, E_1^{-\mathcal{B}}) = \begin{cases} (D, D) & \text{if } R^+ \cap R^- \neq \emptyset \\ (R^+, R^{+c}) & \text{if } R^+ \cap R^- = \emptyset, I^{\mathcal{B}} \cup \neg I^{\mathcal{B}c} \cup R^+ \cup \neg R^- \models \phi \\ (R^+, R^-) & \text{if } R^+ \cap R^- = \emptyset, I^{\mathcal{B}} \cup \neg I^{\mathcal{B}c} \cup R^+ \cup \neg R^- \not\models \phi \end{cases},$$

where $D = B^n$, n is the arity of E^+, and (R^+, R^-) is the result of unit propagation in ASP [31] on ϕ under $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}c} \cup E^{+\mathcal{B}} \cup \neg E^{-\mathcal{B}}$.

Example 26 (Planning with Cumulative Scheduling Continued)
Continuing our running example, assume that $E^{+^{\mathcal{B}}} = E^{-^{\mathcal{B}}} = \emptyset$. *An observant reader can notice that in all models of* ASP_ϕ, *we should have that* $do(t_4)$ *should be true. This is because if* $do(t_4)$ *is false then* G *becomes not reachable. Our module* ASP_ϕ *can also deduce this fact using its unit-propagation. Therefore,* $do(t_4)$ *belongs to* $E_1^{+^{\mathcal{B}}}$.

Similarly, the module CP_ψ is defined as the set of structures $\mathcal{B}$ such that:

$$(E^{+^{\mathcal{B}}}, E^{-^{\mathcal{B}}}) = \begin{cases} (D, D) & \text{if } R^+ \cap R^- \neq \emptyset \\ (D, D) & \text{if } R^+ \cap R^- = \emptyset, F \text{ is inconsistent with } \psi \\ (R^+, R^-) & \text{if } R^+ \cap R^- = \emptyset, F \text{ is consistent with } \psi \end{cases},$$

where D is as before and $(R^+,\ R^-)$ is the result of constraint propagation on ψ under $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup E_2^{+^{\mathcal{B}}} \cup \neg E_2^{-^{\mathcal{B}}}$, and $F = I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup R^+ \cup \neg R^-$. In practical Constraint Programming solvers, various constraint propagation techniques such as arc-consistency checking and k-consistency checking, are applied and it can be shown that they are all Valid Acceptance Procedures. The reader is referred to [50] for complete details on different propagation techniques.

Example 27 (Planning with Cumulative Scheduling Continued)
Continuing our running example, now assume that $E_2^{+^{\mathcal{B}}}$ *contains* $do(t_4)$ *as discussed in Example 26. Then, for the* CP_ψ *to accept structure* $\mathcal{B}$, *we should have that* $do(t_3)$ *should be false. This is because by the constraint propagation for the cumulative constraint in our example, the* CP_ψ *module understands that* $do(t_3)$ *and* $do(t_4)$ *cannot be true together. Therefore,* $do(t_3)$ *belongs to* $E^{-^{\mathcal{B}}}$.

The compound module $CASP_{\phi\wedge\psi}$ is defined as:

$$CASP_{\phi\wedge\psi} := \pi_{\{I,E\}}(((CP_\psi \rhd ASP_\phi)[E_1^+ = E_2^+][E_1^- = E_2^-]) \rhd TOTAL).$$

The correctness of the module $CASP_{\phi\wedge\psi}$ can be proved using the same arguments as the one in Section 4.4.1.

As a CDNL-like technique is also used in SMT solvers, the above algorithm is modelled similarly to Section 4.4.1. The solver S is defined as a DPLL-based online SAT solver and the module ASP_ϕ (resp. CP_ψ) is associated with an oracle O_{ASP} (resp. O_{CP}). The constructions of these oracles are very similar to the ones described in Section 4.4.1.

Example 28 (Planning with Cumulative Scheduling Continued (O_{ASP} and O_{CP})) *Let ϕ and ψ in $CASP_{\phi\wedge\psi}$ be, respectively, the ASP part and the CP part of the specification in Equation 4.9. Let the structure $\mathcal{B}$ contain the same set of partial assignment as the one in Example 25, i.e., $do_1^{+\mathcal{B}} = \{t_3, t_4\}$, $reaches_1^{+\mathcal{B}} = \{S, V\}$, and $C_1^{+\mathcal{B}} = \top$. When O_{CP} is queried on $\mathcal{B}$, it returns the advice $C_1^+ \wedge do_1^+(t_3) \supset do^-(t_4)$ to the solver S. Obtaining this advice and the advice $do^-(t_4) \supset do_1^-(t_4)$ from O_{ASP}, in the next phase, the solver S will make $do_1^-(t_4)$ true and O_{TOTAL} rejects the new structure from S with the reason $do_1^-(t_4) \supset \neg do_1^+(t_4)$.*

Next, we give exact constructions for the solver S and oracle O_{CP}:

Solver S is a DPLL-based SAT solver (clearly complete and online).

Oracle O_{CP} accepts a partial structure $\mathcal{B}$ iff it does not falsify the constraints described above for module CP_ψ on I, E^+, E^-, E_2^+, and E_2^-. Let (R^+, R^-) denote the result of the constraint propagation on ψ under $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup E_2^{+\mathcal{B}} \cup \neg E_2^{-\mathcal{B}}$. Then, if $\mathcal{B}$ is rejected,

1. If $R^+ \cap R^- \neq \emptyset$ or $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup R^+ \cup \neg R^-$ is inconsistent with ψ, O_{CP} returns a reason ω of the form $\bigwedge_{d\in D_1} E_2^+(d) \wedge \bigwedge_{d\in D_2} E_2^-(d) \supset \bigwedge_{d\in D_3}(E^+(d) \wedge E^-(d))$ with $D_1 \subseteq D$, $D_2 \subseteq D$, $\emptyset \subsetneq D_3 \subseteq D$, $\bigvee_{d\in D_1} \neg l(d) \vee \bigvee_{d\in D_2} l(d)$ is always true in ψ, $\mathcal{B} \models \neg\omega$, where $l(d)$ denotes the atomic formula l in ψ whose associated propositional atom is d. This corresponds to the nogood (the set of literals on the left hand side of the implication of ω which cannot be true together) returned by the conflict analysis mechanism of the CP solver.

2. Otherwise, O_{CP} returns a reason ω of the form $\bigwedge_{d\in D_1} E_2^+(d) \wedge \bigwedge_{d\in D_2} E_2^-(d) \supset \bigwedge_{d\in R^+} E^+(d) \wedge \bigwedge_{d\in R^-} E^-(d)$, where $D_1 \subseteq D, D_2 \subseteq D, \mathcal{B} \models \neg\omega$.

By the definition of CP_ψ, we know that $\mathcal{B}$ falsifies the reason and all models of CP_ψ satisfy the reason. Thus, O_{CP} is complete and constructive. O_{CP} may also return some advices in the same form as any ω above such that $\mathcal{B}$ satisfies the left hand side of the implication, but not the right hand side. Also, since the outputs of CP_ψ always subsume the inputs, O_{CP} may also return the set $\{E_2^+(d) \supset E^+(d) \mid d \in D, \mathcal{B} \models E_2^+(d), \mathcal{B} \not\models E^+(d)\} \cup \{E_2^-(d) \supset E^-(d) \mid d \in D, \mathcal{B} \models E_2^-(d), \mathcal{B} \not\models E^-(d)\}$ as the set of advices. Clearly, all the structures in CP_ψ satisfy all sets of advices. Hence, O_{CP} is an advising oracle. Finally, O_{CP} always makes the correct decision for a total structure and rejects a partial structure only when it falsifies the constraints for CP_ψ. O_{CP} never rejects any good partial

structure $\mathcal{B}$ (although it may accept some bad non-total structures). Therefore, O_{CP} is a verifying oracle.

Proposition 5 *1. Modular system $CASP_{\phi\wedge\psi}$ is the set of structures $\mathcal{B}$ such that $\mathcal{B} \models \phi$ and $\mathcal{B}$ is consistent with ψ according to corresponding theory of the constraints.*

2. Solver S is complete and online.

3. O_{ASP}, O_{CP} and O_{TOTAL} are CCAV oracles.

4. Algorithm 4 on modular system $CASP_{\phi\wedge\psi}$ associated with oracles O_{ASP}, O_{CP}, O_{TOTAL}, and the solver S models the solving procedure of the CASP solver on input formula $\phi \wedge \psi$.

4.5 Related Work

There are many papers on modularity in declarative programming, we only review the most relevant ones. The authors of [36] proposed a multi-language framework for constraint modelling. That work was the initial inspiration of [53], but the authors extended the ideas significantly by developing a model-theoretic framework and introducing a feedback operator that adds a significant expressive power.

An early work on adding modularity to logic programs is [22]. The authors derive a semantics for modular logic programs by viewing a logic program as a generalized quantifier. The ideas are further generalized in [46] by considering the concept of modules in declarative programming and introducing modular equivalence in normal logic programs under the stable model semantics. This line of work is continued in [35] to define modularity for disjunctive logic programs. There are also other approaches to adding modularity to ASP languages [13, 33, 8, 7]. The related approach of ID-Logic is described in [16].

The works mentioned earlier focus on the theory of modularity in declarative languages. However, there are also papers that focus on the practice of modular declarative programming and, in particular, solving. These generally fall into one of the two following categories. The first category consists of practical modelling languages which incorporate other modelling languages. For example, X-ASP [52] and ASP-PROLOG [23] extend Prolog with ASP, CP techniques are incorporated into ASP solving in [9], [40], and [30]. Also, ESRA

[25], ESSENCE [26] and Zinc [14] are CP languages extended with features from other languages. However, these approaches give priority to the host language while our modular setting gives equal weight to all modelling languages that are involved. It is important to note that, even in the presence of this distinction, such works have been very important in the development of this chapter because they provide guidelines on how a practical solver deals with efficiency issues. The second category is related to multi-context systems. In [11], the authors introduce non-monotonic bridge rules to the contextual reasoning and originated an interesting and active line of research followed by many others for solving or explaining inconsistencies in non-monotonic multi-context systems [6, 10, 21, 20]. However, these papers do not consider the model expansion task. Moreover, the motivations of these works originate from distributed or partial knowledge, e.g., when agents interact or when trust or privacy issues are important. Despite these differences, the field of multi-context systems is very relevant to our research. Investigating this connection is an important future research direction.

4.6 Conclusion

We took a language-independent view on iterative modular problem solving. Our algorithm is designed to solve combinatorial search problems described as modular systems in the context of model expansion. This model-theoretic approach allows us to abstract away from particular languages of the modules. We performed several case studies of our algorithm in relation to existing systems such as DPLL(T), ILP, ASP-CP. We demonstrated that, in the context of the model expansion task, our algorithm generalizes the work of these solvers. As a side effect of this analysis, we demonstrated how Valid Acceptance Procedures from different communities could be used to implement oracles for modules to achieve efficient solving. For example, the procedures of Well-Founded Model computation and Arc-Consistency checking can be used to implement oracles for the ASP and CP languages to construct an efficient combined solver, which corresponds to the state-of-the-art combination of ASP and CP described in [30].

Our general approach for solving modular systems can be applied to systems such as Business Process Planners in different areas and their variants including Logistics Service

Provider, Manufacturer Supply Chain Management, Mid-size Businesses Relying on External Web Services and Cloud Computing. With the increasing use of service-oriented architecture, such modular systems will become increasingly more applicable. We believe we are taking important initial steps addressing the core aspect of this complex multi-dimensional problem, namely the underlying computationally complex task. As a future direction, we plan to develop a prototype implementation of our algorithms. This thesis is a continuation of [53, 57, 58].

Chapter 5

Conclusion

This thesis contains two different parts. In the first part, a model-based solver Enfragmo is presented. Its language supports full first order logic extended with a limited form of inductive definitions, arithmetic operations, and aggregates. Despite its rich language, Enfragmo's performance is comparable to the state-of-the-art model based solvers as demonstrated through experiments. In the second part, modularity concepts are added to the MX based framework. The extended framework allows one to combine individual modules in a model-theoretic level, independent of the specific languages used to describe each module. An abstract algorithm is introduced to search for a structure in a given compound modular system, and through several case studies, the algorithm is shown to be closely related to the work done in different research areas such as Satisfiability Modulo Theories, Integer Linear Programming, Satisfiability Testing, Constraint Programming, and Answer Set Programming.

Appendix A

Syntax of Enfragmo System

A.1 Problem Specification Grammar

```
<theory_file> ::= <given_part> <find_part>
                                       <phase_part> <print_part>

<given_part> ::= GIVEN : <types_decl> ; <funcs_decl> ;
           | GIVEN : <types_decl> ; <preds_decl> ;
           | GIVEN : <types_decl> ; <preds_decl> ;
                                                  <funcs_decl> ;

<types_decl> ::= TYPES : <identifier_list> ;
                                    INTTYPES : <identifier_list>
           | TYPES : <identifier_list>
               | INTTYPES : <identifier_list>

<identifier_list> ::= | <identifier_list> <identifier>

<preds_decl> ::= PREDICATES : <preds_list>

<a_pred_DCL> ::= <identifier> (<IdentifierListSeparatedByComma>)

<preds_list> ::= | <a_pred_DCL> <preds_list>

<IdentifierListSeparatedByComma> ::= | <identifier>
```

```
               | <IdentifierListSeparatedByComma> , <identifier>

<funcs_decl> ::= FUNCTIONS : <funcs_list>

<funcs_list> ::= | <func_DCL> <funcs_list>

<func_DCL> ::=   <identifier> ( ) : <identifier>
 | <identifier> (<IdentifierListSeparatedByComma>): <identifier>

<find_part> ::= FIND : <identifier_list> ;

<phase_part> ::= <a_phase> | <a_phase>  <phase_part>

<a_phase> ::= PHASE : <fixpoint_part> <satisfying_part>

<fixpoint_part> ::= | FIXPOINT ( <identifier_list> ) :
                                              <induction_part> ;

<induction_part> ::= <an_induction>
                               | <induction_part> <an_induction>

<an_induction> ::= <an_inflation> | <a_definition>

<an_inflation> ::= INFLATE <inflate_description>

<an_inflate_description> ::= { <var_DCL> : <identifier>
         ( <IdentifierListSeparatedByComma> ) <=> <FO_formula> }

<inflate_description> ::= <an_inflate_description>
                | <inflate_description> <an_inflate_description>

<a_definition> ::= DEFINE { <induction_description> }

<induction_description> ::= <an_induction_description>
            | <induction_description> <an_induction_description>

<an_induction_description> ::= <var_DCL> : <identifier>
```

```
            ( <IdentifierListSeparatedByComma> ) <- <FO_formula>

<satisfying_part> ::= | SATISFYING : <satisfying_rules>

<satisfying_rules> ::= <FO_formula> ;
                              | <satisfying_rules> <FO_formula> ;

<FO_formula> ::= ( <FO_formula> ) | <unitary_formula>
           | <FO_formula> <connective> <unitary_formula>

<unitary_formula> ::= ( <FO_formula> )
         | <quantifier> <var_DCL> : <FO_formula>
         | <quantifier> <a_var_DCL> <ord_operator>
                                 <term_nodes>: <unitary_formula>
         | <unitary_formula> <binary_operator> <unitary_formula>
         | ~ <unitary_formula>
         | <atomic_formula>

<atomic_formula> ::= <relation_formula>
         | SUCC [ <identifier> ] ( <term_nodes> , <term_nodes> )
         | <ord_relation> | TRUE | FALSE

<relation_formula> ::= <identifier> ( <args> )

<ord_relation> ::= <term_nodes> <ord_operator> <term_nodes>

<min_func> ::= MIN [ <identifier> ]

<max_func> ::= MAX [ <identifier> ]

<size_func> ::= SIZE [ <identifier> ]

<abs_func> ::= ABS ( <term_nodes> )

<func_ref> ::= <identifier> ( <args> ) | <identifier> ( )

<var_DCL> ::= | <a_var_DCL> | <var_DCL> <a_var_DCL>
```

```
<a_var_DCL> ::= <identifier> : <identifier>

<args> ::= | <term_nodes> | <args> , <term_nodes>

<term_nodes> ::= <a_term_node> | ( <term_nodes> )
               | <term_nodes> + <term_nodes>
               | <term_nodes> * <term_nodes>
               | <term_nodes> - <term_nodes>

<a_term_node> ::= <var_ref> | <min_func> | <max_func>
                | <abs_func> | <func_ref> | <size_func>
                | <aggregate> | <int_term_node>

<aggregate> ::= COUNT { <var_DCL> ; <FO_formula> }
| MIN { <var_DCL> ; <term_nodes> ; <FO_formula> ; <term_nodes> }
| MAX { <var_DCL> ; <term_nodes> ; <FO_formula> ; <term_nodes> }
| SUM { <var_DCL> ; <term_nodes> ; <FO_formula> }

<var_ref> ::= <identifier>

<int_term_node> ::= <int_number> | INTEGER { <term_nodes> }

<arit_operator> ::= + | * | -

<quant_part> ::= <quantifier> <var_DCL>;

<quantifier> ::= ? | !

<binary_operator> ::= & | '|' (or)

<ord_operator> ::= < | <= | > | >= | =

<connective> ::= & | '|' (or) | => | <=>

<print_part> ::= | PRINT : <predicates>
```

```
<predicates> ::= | <predicates> <identifier>

<identifier> ::= [a-zA-Z][0-9a-zA-Z_]+
```

A.2 Instance Description Grammar

```
<instance_file> ::= <type_parts> <pred_parts> <func_parts>

<type_parts> ::= <a_type_part> | <type_parts> <a_type_part>

<a_type_part> ::= TYPE <identifier> <range>

<range> ::= [ continous_values ]

<continous_values> ::= <integer>..<integer>

<discrete_values> ::= <a_discrete_value>
                    | <discrete_values>, <a_discrete_value>

<a_discrete_value> ::= <integer> | <string>

<pred_parts> ::= | <pred_parts> <a_predicate_part>

<a_predicate_part> ::= PREDICATE <identifier> <predicate_values>

<predicate_values> ::= | <a_predicate_value>
                     | <predicate_values> <a_predicate_value>

<a_predicate_value> ::= (<discrete_values>)

<func_parts> ::= | <func_parts> <a_function_part>

<a_function_part> ::= FUNCTION <identifier> <function_values>

<function_values> ::= <a_function_value>
```

```
                    | <function_values> <a_function_value>

<a_function_value> ::= (<func_args>:<func_return_value>)

<func_args> ::= | <discrete_values>

<func_return_value> ::= <integer> | <string>

<identifier> ::= [a-zA-Z][0-9a-zA-Z_]+

<string> ::= '[0-9a-zA-Z_]+'

<integer> ::= [0-9]+
```

Bibliography

[1] http://www.cs.sfu.ca/research/groups/mxp/.

[2] Amir Aavani, Shahab Tasharrofi, Gulay Ünel, Eugenia Ternovska, and David G. Mitchell. Speed-up techniques for negation in grounding. In *LPAR-16*, pages 13–26, 2010.

[3] Amir Aavani, Xiongnan (Newman) Wu, David G. Mitchell, and Eugenia Ternovska. Grounding Cardinality Constraints. *LPAR-16 short paper*, 2010.

[4] Amir Aavani, Xiongnan (Newman) Wu, Shahab Tasharrofi, Eugenia Ternovska, and David G. Mitchell. Enfragmo: A system for modelling and solving search problems with logic. In *18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 15–22, 2012.

[5] Amir Aavani, Xiongnan (Newman) Wu, Eugenia Ternovska, and David G. Mitchell. Grounding formulas with complex terms. In *Canadian AI, the 24th Canadian Conference on Artificial Intelligence*, pages 13–25, 2011.

[6] Seif El-Din Bairakdar, Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. The dmcs solver for distributed nonmonotonic multi-context systems. In Tomi Janhunen and Ilkka Niemelä, editors, *Logics in Artificial Intelligence - 12th European Conference, JELIA 2010, Helsinki, Finland, September 13-15, 2010. Proceedings*, volume 6341 of *Lecture Notes in Computer Science*, pages 352–355. Springer, 2010.

[7] Marcello Balduccini. Modules and signature declarations for a-prolog: Progress report. In *SEA*, pages 41–55, 2007.

[8] Chitta Baral, Juraj Dzifcak, and Hiro Takahashi. Macros, macro calls and use of ensembles in modular answer set programming. In *ICLP'06. LNCS*, pages 376–390, 2006.

[9] Sabrina Baselice, Piero A. Bonatti, and Michael Gelfond. Towards an integration of answer set and constraint solving. In *In Proc. of ICLP'05*, pages 52–66, 2005.

[10] Markus Bögl, Thomas Eiter, Michael Fink, and Peter Schüller. The mcs-ie system for explaining inconsistency in multi-context systems. In Tomi Janhunen and Ilkka Niemelä, editors, *Logics in Artificial Intelligence - 12th European Conference, JELIA 2010, Helsinki, Finland, September 13-15, 2010. Proceedings*, volume 6341 of *Lecture Notes in Computer Science*, pages 356–359. Springer, 2010.

[11] Gerhard Brewka and Thomas Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 1*, pages 385–390. AAAI Press, 2007.

[12] Broes De Cat and Marc Denecker. DPLL(Agg): An efficient SMT module for aggregates. In *LaSh'10 Workshop*, 2010.

[13] Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Modular nonmonotonic logic programming revisited. In Hill and Warren [33], pages 145–159.

[14] Maria J. García de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace. The modelling language Zinc. *Principles and Practice of Constraint Programming-CP 2006*, pages 700–705, 2006.

[15] Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Christoph Koch, Nicola Leone, Simona Perri, and Gerald Pfeifer. System description: DLV. In *LPNMR*, pages 424–428, 2001.

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

[17] Marc Denecker and Eugenia Ternovska. A logic of non-monotone inductive definitions. *TOCL*, (2):1–51, 2008.

[18] Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Miroslaw Truszczyński. The second answer set programming competition. *LPNMR*, pages 637–654, 2009.

[19] Niklas Een and Niklas Sörensson. MiniSat v1.13 - A SAT Solver with Conflict-Clause Minimization, System description for the SAT competition, 2005.

[20] Thomas Eiter, Michael Fink, and Peter Schüller. Approximations for explanations of inconsistency in partially known multi-context systems. In James P. Delgrande and Wolfgang Faber, editors, *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, volume 6645 of *Lecture Notes in Computer Science*, pages 107–119. Springer, 2011.

[21] Thomas Eiter, Michael Fink, Peter Schüller, and Antonius Weinzierl. Finding explanations of inconsistency in multi-context systems. In Fangzhen Lin, Ulrike Sattler, and

Miroslaw Truszczynski, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*. AAAI Press, 2010.

[22] Thomas Eiter, Georg Gottlob, and Helmut Veith. Modular logic programming and generalized quantifiers. In *Proc. of LPNMR*, pages 290–309. Springer-Verlag, 1997.

[23] Omar Elkhatib, Enrico Pontelli, and Tran Cao Son. ASP- PROLOG: A System for Reasoning about Answer Set Programs in Prolog. In *the Proc. of PADL'04*, pages 148–162, 2004.

[24] Ronald Fagin. Generalized first-order spectra and polynomial-time recognizable sets. *Complexity of Computation*, pages 43–74, 1974.

[25] Pierre Flener, Justin Pearson, and Magnus Agren. Introducing ESRA, a relational language for modelling combinatorial problems. In *Proc., LOPSTR'03*, 2003.

[26] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13:268–306, 2008.

[27] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The Potsdam answer set solving collection. *AI Commun. 24(2)*, pages 105–124, 2011.

[28] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *Proceedings of the 20th international joint conference on Artifical intelligence*, IJCAI'07, pages 386–392, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[29] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *IJCAI*, pages 386–392, 2007.

[30] Martin Gebser, Max Ostrowski, and Torsten Schaub. Constraint answer set solving. In *Proc. of ICLP'09*, LNCS, pages 235–249. Springer-Verlag, 2009.

[31] Martin Gebser and Torsten Schaub. Characterizing asp inferences by unit propagation. In *IN: LASH ICLP WORKSHOP*, pages 41–56, 2006.

[32] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.

[33] Patricia M. Hill and David Scott Warren, editors. *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, volume 5649 of *Lecture Notes in Computer Science*. Springer, 2009.

[34] Wilfrid Hodges. Cambridge University Press, 1993.

[35] Tomi Janhunen, Emilia Oikarinen, Hans Tompits, and Stefan Woltran. Modularity aspects of disjunctive stable models. In *Proc. of LPNMR*, volume 4483 of *LNAI*, pages 175–187, 2007.

[36] Matti Järvisalo, Emilia Oikarinen, Tomi Janhunen, and Ilkka Niemelä. A module-based framework for multi-language constraint modeling. In *Proc. of LPNMR*, volume 5753 of *LNCS*, pages 155–168. Springer-Verlag, 2009.

[37] Phokion G. Kolaitis and Christos H. Papadimitriou. Why not negation by fixpoint? In *Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '88, pages 231–239, New York, NY, USA, 1988. ACM.

[38] Maarten Mariën, Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. Sat(id): Satisfiability of propositional logic extended with inductive definitions. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing SAT 2008*, volume 4996 of *Lecture Notes in Computer Science*, chapter 20, pages 211–224. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[39] Joaäo P. Marques-silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[40] Veena S. Mellarkod, Michael Gelfond, and Yuanlin Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53:251–287, 2008.

[41] David G. Mitchell. A sat solver primer. *Bulletin of the EATCS*, 85:112–132, 2005.

[42] David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In *Proc. AAAI*, pages 430–435, 2005.

[43] Raheleh Mohebali. A method for solving NP search problems based on model expansion and grounding. Master's thesis, Simon Fraser University, 2006.

[44] Ilkka Niemelä. Integrating answer set programming and satisfiability modulo theories. In *LPNMR*, volume 5753 of *LNCS*, pages 3–3. Springer-Verlag, 2009.

[45] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53:937–977, November 2006.

[46] Emilia Oikarinen and Tomi Janhunen. Modular equivalence for normal logic programs. In *the Proc. of NMR'06*, pages 10–18, 2006.

[47] Panos M. Pardalos and Mauricio G. C. Resende. *Handbook of applied optimization*, volume 126. Oxford University Press New York;, 2002.

[48] Murray Patterson, Yongmei Liu, Eugenia Ternovska, and Arvind Gupta. Grounding for model expansion in k-guarded formulas with inductive definitions. In *Proc. IJCAI'07*, pages 161–166, 2007.

[49] Francesco Ricca and Nicola Leone. Disjunctive logic programming with types and objects: The dlv+ system. *Journal of Applied Logic*, 5(3):545 – 573, 2007.

[50] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[51] Roberto Sebastiani. Lazy Satisfiability Modulo Theories. *JSAT*, 3:141–224, 2007.

[52] Terrance Swift and David S. Warren. *The XSB System*, 2009.

[53] S. Tasharrofi and E. Ternovska. A semantic account for modularity in multi-language modelling of search problems. In *8th International Symposium Frontiers of Combining Systems (FroCoS)*, October 2011.

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

[55] Shahab Tasharrofi and Eugenia Ternovska. PBINT, a logic for modelling search problems involving arithmetic. In *Proc. LPAR-17*. Springer, October 2010. LNCS 6397.

[56] Shahab Tasharrofi, Xiongnan Newman Wu, and Eugenia Ternovska. Solving modular model expansion tasks. In *25th Workshop on Logic Programming*, September 2011.

[57] Shahab Tasharrofi, Xiongnan (Newman) Wu, and Eugenia Ternovska. Solving modular model expansion tasks. *CoRR*, abs/1109.0583, 2011. 25'th Workshop on Logic Programming.

[58] Shahab Tasharrofi, Xiongnan Newman Wu, and Eugenia Ternovska. Solving modular model expansion: Case studies. In *INAP/WLP*, July 2012.

[59] Eugenia Ternovska and David G. Mitchell. Declarative programming of search problems with built-in arithmetic. In *Proc. of IJCAI*, pages 942–947, 2009.

[60] Emina Torlak. *A constraint solver for software engineering: finding models and cores of large relational specifications*. PhD thesis, Cambridge, MA, USA, 2009.

[61] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer Berlin / Heidelberg, 2007.

[62] Grigori S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, pages 115–125, 1968.

[63] Pashootan Vaezipoor, David Mitchell, and Maarten Mariën. Lifted unit propagation for effective grounding. *CoRR*, abs/1109.1317, 2011. Appears in the Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011).

[64] Maarten H. Van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, October 1976.

[65] Allen Van Gelder. The alternating fixpoint of logic programs with negation. *J. Comput. Syst. Sci.*, 47(1):185–221, August 1993.

[66] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38:619–649, July 1991.

[67] Johan Wittocx. Finite domain and symbolic inference methods for extensions of first-order logic. *AI Commun.*, 24(1):91–93, January 2011.

[68] Johan Wittocx, Maarten Marién, and Marc Denecker. The IDP system: A model expansion system for an extension of classical logic. In *Proceedings of the 2nd Workshop on Logic and Search*, pages 153–165, 2008.

[69] Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding fo and fo(id) with bounds. *J. Artif. Int. Res.*, 38(1):223–269, May 2010.