ARITHMETIC AND MODULARITY IN DECLARATIVE LANGUAGES FOR KNOWLEDGE REPRESENTATION

by

Shahab Tasharrofi

M.Eng., Sharif University of Technology, 2008

B.Eng., Iran University of Science and Technology, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Fall 2013

APPROVAL

Name:	Shahab Tasharrofi
Degree:	Doctor of Philosophy
Title of Thesis:	Arithmetic and Modularity in Declarative Languages for Knowledge Representation
Examining Committee:	Dr. Andrei Bulatov, Associate Professor, Computing Science, Simon Fraser University Chair
	Dr. Eugenia Ternovska, Associate Professor, Computing Science, Simon Fraser University Senior Supervisor
	Dr. David G. Mitchell, Associate Professor, Computing Science, Simon Fraser University Supervisor
	Dr. Uwe Glässer, Professor, Computing Science, Simon Fraser University Supervisor
	Dr. Oliver Schulte, Associate Professor, Computing Science, Simon Fraser University SFU Examiner
	Dr. Miroslaw Truszczynski, External Examiner, Professor of Computer Science, University of Kentucky
Date Approved:	December 16, 2013

Partial Copyright Licence

Abstract

The past decade has witnessed the development of many important declarative languages for knowledge representation and reasoning such as answer set programming (ASP) languages and languages that extend first-order logic. Also, since these languages depend on background solvers, the recent advancements in the efficiency of solvers has positively affected the usability of such languages. This thesis studies extensions of knowledge representation (KR) languages with arithmetical operators and methods to combine different KR languages.

With respect to arithmetic in declarative KR languages, we show that existing KR languages suffer from a huge disparity between their expressiveness and their computational power. Therefore, we develop an ideal KR language that captures the complexity class NP for arithmetical search problems and guarantees universality and efficiency for solving such problems.

Moreover, we introduce a framework to language-independently combine modules from different KR languages. We study complexity and expressiveness of our framework and develop algorithms to solve modular systems. We define two semantics for modular systems based on (1) a model-theoretical view and (2) an operational view on modular systems. We prove that our two semantics coincide and also develop mechanisms to approximate answers to modular systems using the operational view. We augment our algorithm these approximation mechanisms to speed up the process of solving modular system.

We further generalize our modular framework with supported model semantics that disallows self-justifying models. We show that supported model semantics generalizes our two previous model-theoretical and operational semantics. We compare and contrast the expressiveness of our framework under supported model semantics with another framework for interlinking knowledge bases, i.e., multi-context systems, and prove that supported model semantics generalizes and unifies different semantics of multi-context systems. Motivated by the wide expressiveness of supported models, we also define a new supported equilibrium semantics for multi-context systems and show

that supported equilibrium semantics generalizes previous semantics for multi-context systems. Furthermore, we also define supported semantics for propositional programs and show that supported model semnatics generalizes the acclaimed stable model semantics and extends the two celebrated properties of rationality and minimality of intended models beyond the scope of logic programs.

Keywords: Knowledge Representation and Reasoning, Declarative Problem Solving, Built-in Arithmetic, Modularity, Language-independence, Supported Semantics, Stable Model Semantics, Multi-context Systems

To my beautiful wife, Hanie, for all her emotional support

"Pure mathematics consists entirely of assertions to the effect that, if such and such a proposition is true of anything, then such and such another proposition is true of that thing. It is essential not to discuss whether the first proposition is really true, and not to mention what the anything is, of which it is supposed to be true ... If our hypothesis is about anything, and not about some one or more particular things, then our deductions constitute mathematics. Thus mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true. People who have been puzzled by the beginnings of mathematics will, I hope, find comfort in this definition, and will probably agree that it is accurate."

— Principles of Mathematics, International Monthly, vol. 4, BERTRAND RUSSELL, *1901*

Acknowledgments

First and foremost, I would like to offer my utmost gratitude towards my senior supervisor, Dr. Eugenia Ternovska for guiding me and supporting me with her insightful vision. Not only has Eugenia always been a source of inspiration, but, also, she has instilled in me the two most essential skills for a researcher: finding worthwhile questions and setting your goals correctly.

I would also like to express my gratitude towards the other two members of my PhD committee: Dr. David Mitchell and Dr. Uwe Glässer. David has always been full of interesting ideas and Uwe has helped me discover interesting related works in the field of Software Engineering.

Moreover, I feel obliged to extend my sincerest gratitude towards the two examiners of my PhD research: Dr. Oliver Schulte, the internal examiner and Dr. Miroslaw Truszczynski, my external examiner for their helpful comments on my work.

Furthermore, I want to thank my many collaborators during these years that include: Xiongnan (Newman) Wu, Amir Aavani, Pashootan Vaezipoor, Alireza Ensan, Maarten Marien, etc. Without them, many of the discussions that directly affected this thesis would have simply not existed.

Last but not the least, I would like to thank my family and my wife for believing in me and helping me in whatever way possible through these years.

Shahab Tasharrofi

Contents

List of Figures

Chapter 1

Introduction

Declarative programming is a branch of programming that is built on the idea of describing what we need to do instead of how we need to do it. Traditionally, declarative programming has been considered as an alternative to imperative or functional programming and it has mainly been used in the form of query languages for databases. However, the close relation between declarative programming paradigms, fragments of logic, and, the ability to interpret logical statements as specification of computation, has resulted in its wide adoption by many sub-communities of computing science as their paradigm of choice for the free of side-effect manipulation of knowledge. One of the main such communities that has adopted declarative programming is the community of artificial intelligence (AI) researchers.

The fundamental goal of Artificial Intelligence is *to simulate human behavior*. There are many questions that need to be answered with respect to this broad and general goal such as how to receive audio, video, textual, or other form of input from the environment, how to understand the contexts of such inputs, how to effectively reason about those inputs, and, how to manipulate the environment in response to those inputs.

Since the main goal of logic is to study reasoning procedures, it is no surprise that *computational logic* has long been the chosen method to study and implement the (explicit) reasoning procedures that are needed in AI. That is, once the input is received from the environment and it is known what goals are being pursued by an agent, (explicit) reasoning usually happens in the form of mathematical manipulations of an agent's knowledge of its environment in a language that is also motivated by the environment of the agent.

Knowledge representation and reasoning (KRR) is a branch of AI that is concerned with inventing the appropriate syntax and semantics for succinctly representing an agent's knowledge of its

environment and to effectively reason about how to achieve the agent's goals. Note that, as long as AI community is concerned, the representation of knowledge and reasoning about it are not required to be in any specific form. Indeed, there is no requirement on this knowledge being represented in a human-readable format. For instance, a neural network represents knowledge about its environment in a completely illegible form but still has wide applications. However, the readability and the changeability of knowledge in a logical form and the guarantees logic provides (such as completeness of reasoning procedures with respect to a domain of knowledge) has made formal logic the main approach taken by the KRR community. Thus, declarative programming in general and logic programming in particular are both the main tool and the main subject of research in the KRR community. Among such declarative languages, Prolog [39] is the most well-known language that has cultivated many new research directions in KRR as well as benefited from the results of such research.

During the past decade, KRR community has invented many new declarative programming frameworks that are of limited expressiveness (i.e., some problems cannot be expressed in these languages) but of better computational performance. Among such languages, one can mention many variants of Answer Set Programming (ASP) [32] (such as the system languages of Gringo [80] and DLV [49]), several variants of first-order logic (e.g., FO(ID) [55] and the system languages of IDP [212] and Enfragmo [1]), as well as many constraint programming languages (such as Essence [76] and Zinc [48]). All these languages have enjoyed wide acceptance due to the existence of efficient solvers for them. For example, the system language of Gringo depends on a fast ASP solver known as Clasp [82]. Another example is the Enfragmo system that is designed to be able to use all efficient satisfiability (SAT) solvers in the background.

Therefore, the recent advancements in the scale of the problems that such propositional solvers (i.e., propositional ASP, SAT, etc. solvers) deal with and the huge improvements in the efficiency of these propositional solvers has had an immensely positive effect on the usability of the declarative programs that depend on them. These advancements and improvements are showcased regularly during SAT and ASP competitions.

The declarative languages above have their own similarities and differences. The similarity between these languages is that they are all solving instances of search problems. For example, SAT solvers look for an assignment that satisfies all their clauses in Conjunctive Normal Form (CNF). Usually, such CNF clauses encode a problem's specification plus its instance and, a satisfying assignment for that CNF is directly convertible to a solution to a specific instance of a problem. Similarly, a propositional answer set program is usually the result of grounding an answer set program with

	7				9	6		3
8	5		6	7	4			2
6				1	3			5
	8	1			2			
	9	2	3	6	1	7	4	
			9			2	3	
2			1	9				4
9			4	3	6		2	7
4		7	5				9	

Figure 1.1: An instance of a Sudoku puzzle with $n = 3$.

variables with respect to a given instance.

The idea of "model expansion" generalizes the task of searching for a solution to an instance of a problem. The model expansion task for a formula ϕ in a logic $\mathcal{L}$ takes a structure $\mathcal{A}$ over vocabulary σ and expands $\mathcal{A}$ into a structure $\mathcal{B}$ over vocabulary $(\sigma \cup \varepsilon)$ (with $\sigma \cap \varepsilon = \emptyset$) such that $\mathcal{B}$ agrees with $\mathcal{A}$ on its domain and interpretation of vocabulary symbols in σ (i.e., $dom(\mathcal{A}) = dom(\mathcal{B})$ and $R^{\mathcal{A}} = R^{\mathcal{B}}$ for all $R \in \sigma$), and, $\mathcal{B}$ is a model of formula ϕ in logic $\mathcal{L}$.

To make things clearer, in the context of model expansion, the formula ϕ represents the problem that needs to be solved. The structure $\mathcal{A}$ represents a specific instance of that problem and the interpretations provided by structure $\mathcal{B}$ for symbols in ε are known as solutions to the instance represented by $\mathcal{A}$. The task of model expansion, itself, represents the procedure of searching for the right solution $\mathcal{B}$ for instance $\mathcal{A}$ of problem ϕ. This essential similarity is emphasized by Mitchell and Ternovska in [149]. Following them, we also use model expansion in this thesis as the task that underlies search problems.

Example 1.1 (Sudoku Puzzle: Model Expansion) *In the general version of Sudoku puzzle, you are given a number n and an $n^2 \times n^2$ grid with some of its cells filled with numbers $1, \ldots, n^2$. You are asked to fill the rest of grid with, again, numbers $1, \ldots, n^2$ so that all numbers in a row, in a column, and in smaller $n \times n$ sub-grids are different. Each such sub-grid starts at position $(k_1 \times n+1, k_2 \times n+1)$ and ends at position $(k_1 \times n+n, k_2 \times n+n)$ for some $k_1, k_2 \in \{0, \ldots, n-1\}$. Figure 1.1 shows an instance of the Sudoku puzzle for $n = 3$.*

Here, the instance vocabulary is $\sigma = \{n, f\}$ with f showing the partial filling of the Sudoku

grid. For example, structure $\mathcal{A}$ depicted in Figure 1.1 interprets n by 3 and f by

$$\{(1,2,8),(1,3,6),(1,7,2),(1,8,9),(1,9,4),(2,1,7),(2,2,5),$$
$$(2,4,8),(2,5,9),(3,4,1),(3,5,2),(3,9,7),(4,2,6),(4,5,3),$$
$$(4,6,9),(4,7,1),(4,8,4),(4,9,5),(5,2,7),(5,3,1),(5,5,6),$$
$$(5,7,9),(5,8,3),(6,1,9),(6,2,4),(6,3,3),(6,4,2),(6,5,1),$$
$$(6,8,6),(7,1,6),(7,5,7),(7,6,2),(8,5,4),(8,6,3),(8,8,2),$$
$$(8,9,9),(9,1,3),(9,2,2),(9,3,5),(9,7,4),(9,8,7)\}.$$

Here, the expansion vocabulary ε has function $v : [1\cdots9] \times [1\cdots9] \mapsto [1\cdots9]$ such that v agrees with pre-fillings given by f and also assigns each of the values 1 to 9 exactly once to each row, each column and each smaller grid. Also, $\mathcal{L}$ and ϕ are, respectively, a logic and a formula in that logic so that ϕ is satisfied by an expansion $\mathcal{B}$ of $\mathcal{A}$ if and only if $\mathcal{B}$ interprets v according to the definition of the Sudoku puzzle. Later on, we will give two choices for such ϕ and $\mathcal{L}$.

In our description of the task of model expansion, the role of logic $\mathcal{L}$ and formula ϕ is intertwined, i.e., one of them is meaningless without the other one. Moreover, the role of formula ϕ and logic $\mathcal{L}$ can be replaced by the set of possible models of ϕ in $\mathcal{L}$. Using this replacement, the task of model expansion can be defined independently of the syntax and/or semantics of a logic: Given a class $\mathcal{K}$ of $(\sigma \cup \varepsilon)$-structures (with $\sigma \cap \varepsilon = \emptyset$) and a σ-structure $\mathcal{A}$, find a structure $\mathcal{B} \in \mathcal{K}$ that expands $\mathcal{A}$, i.e., has the same domain as $\mathcal{A}$ and agrees with $\mathcal{A}$ on the interpretations of vocabulary symbols in σ. This definition uses $\mathcal{K}$ to represent problems (instead of ϕ in the previous definition). Thus, it completely adheres to how problems are normally defined in logic, i.e, as a set of structures.

Although all the declarative languages that we mentioned are similar with respect to the task they focus on, i.e., model expansion, they are different on many other aspects, e.g., the basic constructs of their languages as well as the methods that they use for finding solutions. Each of the declarative languages that we mentioned above and their corresponding propositional solvers have their own set of basic constructs, e.g., clauses in SAT and normal/disjunctive rules in ASP. Such restricted core languages facilitated the development of efficient solvers for these declarative languages in their early stages of development. However, under such restrictions, solving a problem with complex constraints using a particular solver requires an encoding of those complex constraints using the basic constructs provided by the language. Such an encoding is problematic on two fronts:

(†1) They hinder users from appreciating declarative languages because encoding (and, more importantly, efficient encodings) are non-trivial and tiresome tasks.

(†2) During the process of encoding, a problem's structure is usually lost. It means that while the solutions to a problem's encoding can be translated back into solutions to the main problems, the methods to obtain a solution and the methods to trim the search space for a solution are not necessarily preserved during the encoding process. Therefore, solvers may have to spend a huge computational overhead to discover facts about a problem's encoding that would have been immediate if the structure of the problem was not lost (e.g., if encoding was not necessary).

As an example of how encoding a problem can be cumbersome and uneffective, consider the Sudoku puzzle from Example 1.1:

Example 1.2 (Sudoku Puzzle: Natural Specification) *Consider the Sudoku puzzle from Example 1.1. The following axioms naturally specify the constraints on a solution of a Sudoku puzzle. So, we are looking for a function* $v : [1, \ldots, n^2] \times [1, \ldots, n^2] \mapsto [1, \ldots, n^2]$ *so that:*

$$\forall x, y, k \in [1, \ldots, n^2] : (f(x, y, k) \supset v(x, y) = k),$$
$$\forall x \in [1, \ldots, n^2] : \mathrm{perm}([v(x, y) \mid y \in [1, \ldots, n^2]], [1, \ldots, n^2]),$$
$$\forall y \in [1, \ldots, n^2] : \mathrm{perm}([v(x, y) \mid x \in [1, \ldots, n^2]], [1, \ldots, n^2]),$$
$$\forall x, y \in [0, \ldots, n-1] : \mathrm{perm}([v(x \times n + x', y \times n + y') \mid x', y' \in [1, \ldots, n]], [1, \ldots, n^2]).$$

where $\mathrm{perm}(L_1, L_2)$ *is true if list* L_1 *is a permutation of list* L_2*. Note that the formulas above use a very extensive set of constructs that are not available in most existing declarative languages. Among such constructs, one could mention arithmetical constructs, (finite) list generators, and (finite) list permutations.*

Example 1.2 shows an intuitive specification of the Sudoku problem. We know that this problem is NP-complete [216] (if the size of input is guaranteed to be a polynomial on n, e.g., if n itself is encoded in unary). Thus, by Fagin's theorem [68], we know that this problem can be encoded in existential second order logic and, thus, also as a model expansion task for first-order (FO) logic [126]. However, the natural encoding we gave in Example 1.2 uses constructs (such as arithmetical constructs, lists and permutations of lists) that are not supported in first-order logic. In fact, the Fagin's theorem and its equivalent for FO-MX (model expansion in FO) only guarantee that *at least one of the encodings* of the Sudoku problem are expressible as FO-MX but *not necessarily its intuitive encoding* (that uses numbers).

Example 1.3 (Sudoku Puzzle: Encoding as a FO-MX Task) *One of the possible encodings of the Sudoku problem is to use a domain of size* n *and encode numbers* $1 \cdots n^2$ *as pairs of elements on*

that domain, i.e., base-n encoding of numbers. For example, the following formulas axiomatize a base-n encoding of the Sudoku problem:

$$
\begin{aligned}
&\forall x_1, x_2, y_1, y_2, k_1, k_2 : (f(x_1, x_2, y_1, y_2, k_1, k_2) \supset v_1(x_1, x_2, y_1, y_2) = k_1),\\
&\forall x_1, x_2, y_1, y_2, k_1, k_2 : (f(x_1, x_2, y_1, y_2, k_1, k_2) \supset v_2(x_1, x_2, y_1, y_2) = k_2),\\
&\forall x_1, x_2, y_1, y_2, y'_1, y'_2 :\\
&\quad ((v_1(x_1, x_2, y_1, y_2) = v_1(x_1, x_2, y'_1, y'_2) \wedge v_2(x_1, x_2, y_1, y_2) = v_2(x_1, x_2, y'_1, y'_2)) \supset\\
&\qquad (y_1 = y'_1 \wedge y_2 = y'_2)),\\
&\forall y_1, y_2, x_1, x_2, x'_1, x'_2 :\\
&\quad ((v_1(x_1, x_2, y_1, y_2) = v_1(x'_1, x'_2, y_1, y_2) \wedge v_2(x_1, x_2, y_1, y_2) = v_2(x'_1, x'_2, y_1, y_2)) \supset\\
&\qquad (x_1 = x'_1 \wedge x_2 = x'_2)),\\
&\forall x_1, x_2, y_1, y_2, x'_2, y'_2 :\\
&\quad ((v_1(x_1, x_2, y_1, y_2) = v_1(x_1, x'_2, y_1, y'_2) \wedge v_2(x_1, x_2, y_1, y_2) = v_2(x_1, x'_2, y_1, y'_2)) \supset\\
&\qquad (x_2 = x'_2 \wedge y_2 = y'_2)),
\end{aligned}
$$

where v_1 and v_2, together, encode v in Example 1.2. It can be easily observed that the encoding above is much less legible, and far harder to axiomatize. This is despite the fact that, here, Sudoku was chosen because we can use base-n encoding to avoid encoding arithmetical operators *such as addition and multiplication. If we had to also encode arithmetical operators, the encoding above would have been far less readable than it is.*

Although legibility is itself a very important hindrance for not-so-experienced users of declarative languages, it is, by no means, the only problem with the second axiomatization of the Sudoku puzzle as given in Example 1.3. That is, even if we could expect users of declarative programming languages to describe their problems (such as the Sudoku problem) in the limited language of first-order logic (as in Example 1.3), there would still exist a very important and inherent problem with such encodings: *the loss of a problem's structure*. In the following example, we continue on Example 1.3 and show that how the second axiomatization of Sudoku puzzle loses the structure of Sudoku puzzle and how this loss of structure affects the solving process for Sudoku.

Example 1.4 (Sudoku Puzzle: Loss of Structure in FO-MX Encoding) *In the axiomatization of the Sudoku puzzle as in Example 1.3, instead of saying that v should take all values of 1 to n^2 in every column, we say that, for each column, v should take different values at different rows (also, similar translation was used for rows and smaller grids). Since the size of all rows, columns and smaller grids are n^2 and since v could take only n^2 different values, the two axiomatizations are*

equivalent. However, the first axiomatization has a very intuitive reading that could have been used to speed up the solving process, i.e., there exists a bijection $g : [1 \cdots n^2] \mapsto [1 \cdots n^2]$ *such that* $v(c, g(k)) = k$ *for all* $k \in [1 \cdots n^2]$.

This intuitive reading is an essential property of the permutation construct used in the natural axiomatization of the Sudoku puzzle. That is, in general, list A *is a permutation of list* B *if and only if (1) they have the same length, i.e.,* $|A| = |B|$*, and, (2) there exists a bijection* $g : \{1, \cdots, |A|\} \mapsto \{1, \cdots, |A|\}$ *such that* $A[p(i)] = B[i]$ *for all* $i \in \{1, \cdots, |A|\}$.

Therefore, one can use a matching algorithm (and Hall's theorem) on the bipartite graph $\mathcal{G} := (\{1, \cdots, n^2\}, \{1, \cdots, n^2\}; \{(i, j) \mid v(c, i) = j \text{ is possible}\})$ *to reduce the set of possible assignments to* v*. For example, if sets* $I, J \subseteq \{1, \cdots, n^2\}$ *exist such that* $|I| \leq |J|$ *and, for all* $i \in I$, $\{j \mid v(c, i) = j \text{ is possible}\} \subseteq J$ *then* $v(c, i') = j$ *becomes impossible for all* $j \in J$ *and* $i' \notin I$*. This method can thus be used to reduce the search space for a solution to an instance of the Sudoku puzzle.*

Of course, one should note that a similar reduction on the set of possible assignments is also possible in the FO-MX axiomatization of the Sudoku puzzle as well. This is because there is an easy bijection between the solutions of the intuitive axiomatization (as in Example 1.2) and the solutions of the FO-MX axiomatization (as in Example 1.3). The difference, however, is that the applicability of matching process above for reducing the set of possible assignments is easily deducible for our natural axiomatization (because it uses permutations that can always benefit from this process) but hard to deduce for the FO-MX axiomatization (because it encodes permutation constructs). Hence, the second axiomatization is said to have lost the structure of the problem in the encoding process.

Broadly speaking, issues (†1) and (†2) can be addressed in two ways: (1) by extending the language of a solver, and, (2) by combining languages of different solvers. We discuss these two directions and our contributions of each of these direction in Sections 1.1 and 1.2.

1.1 Extending the language of a solver

One way to address issues (†1) and (†2) is to extend the language of a solver with new constructs. Such extensions, firstly, relieve users from having to encode their problem using basic constructs and, secondly, better preserve the structure of a problem. Thus, for example, the SAT community has (not surprisingly) massively invested in both devising structure-preserving encodings of complex constraints as well as solver extensions such as native handling of cardinality and pseudo-boolean constraints.

Some of the most important constructs that are missing as a basic construct in many declarative KR languages are the arithmetical operators such as $+$ and $\times$ and arithmetical aggregates such as summation (Σ), product (Π), $\min$ and $\max$. Such arithmetical constructs are so important in the process of axiomatization that they have traditionally been among the first extensions to all KR languages. This is because many important KR applications heavily depend on numbers. For example, the general notion of a plan (that underlies many important AI problems) depends on the existence of successive steps of the plan that are usually represented by natural numbers, e.g., the first step, the second step, etc. Another example is the common notion of a solution's cost that defines a whole area of AI problems known as optimization problems.

Admittedly, both categories of problems above can be defined in terms of objects other than numbers. For example, plans can use all successor relations (whether numerical or not) and cost is definable in terms of general measures (that can be non-numerical). However, certainly, the most intuitive way to define such problems is to use numbers (and, most importantly, natural numbers). It is thus inevitable for a declarative KR language to support arithmetical constructs. This claim can indeed be confirmed by observing that some degree of arithmetical constructs are supported in all KR languages. For example, all the system languages of Enfragmo [1], IDP [212], Gringo [80] and Lparse [182] (the former two extend first-order logic and the latter two extend answer set programs) support basic arithmetical operators for addition, multiplication and subtraction. Moreover, the first three also support arithmetical aggregates to compute cardinality of a set and/or sum, minimum and maximum of a elements in a set.

However, the success of declarative KR languages (in comparison with declarative languages in general) is mostly due to the existence of effective reasoning procedures for declarative KR languages. These efficient reasoning procedures are non-existent for general declarative languages because of the computational complexity associated with the problems expressible in a general language. That is, declarative KR languages have traditionally favored effective computation over expressiveness. This is in contrast to the fact that, by Goëdel's first incompleteness theorem, the unbounded presence of even very basic arithmetical constructs makes a language undecidable. Thus, each KR language has invented its own form of syntactical limitations to guarantee the decidability of its language in the presence of arithmetic. For example, Enfragmo and IDP system languages require numerical quantifiers to use finite guards. Also, ASP languages have, over the years, developed different notions of safety [80], level-restrictedness [86] and ω-restrictedness [79] to guarantee decidability of ASP programs in the presence of unbounded domains (including the domain of integral numbers).

In spite of the wide consensus on the importance of arithmetical constructs in KR languages, its early adoption by those languages, and the effort to keep the complexity of those languages manageable even in the presence of arithmetical constructs, the expressiveness of KR languages in the presence of arithmetical constructs have not yet been studied. Chapter 3 of this thesis addresses this concern. We study the language of several successful KR frameworks with respect to the expressiveness of their arithmetical constructs. All the languages we study, when limited only to the class of finite structures, are known to either express the search problems that are recognizable in the first level of polynomial hierarchy (NP) or express the search problems that are recognizable in the second level of polynomial hierarchy (Σ_2^P). However, once we allow the unbounded domain of natural numbers (or integers), the known expressiveness results for these languages do not apply anymore. We show the non-applicability of previous expressiveness results using two kinds of arguments:

(1) We show that some NP-recognizable and natural arithmetical search problems cannot be expressed in the system language of any of the fore-mentioned declarative KR languages.

(2) We also show that there exist arithmetical search problems that are NEXP-complete and that can be expressed in the language of many declarative KR paradigms including the ones above.

Hence, the addition of arithmetical constructs to these KR languages has hugely increased the difficulty of solving problems in those languages. That is, their complexity has gone from NP to NEXP-complete and, so, their reasoning mechanisms should also become more complex. However, despite the complexity that adding arithmetic has incurred on these languages, they cannot yet express some NP-recognizable arithmetical problems. Such a huge disparity between how computationally complex and how expressive a language becomes in the presence of arithmetical constructs motivated us to design an ideal declarative language whose expressiveness in the presence of arithmetic is controlled.

Thus, we also introduce a guarded logic, known as PBINT, that provably captures the set of arithmetical NP search problems. It means that, we are able to prove that every arithmetical search problem that is recognizable in NP is axiomatizable in PBINT and that every problem that is axiomatizable in PBINT is also recognizable in NP. Hence, PBINT is an example of how a careful design of a declarative languages can, simultaneously, guarantee both efficiency and expressiveness.

Furthermore, we extend PBINT with constructs to handle abstract domain elements. This way, we introduce a multi-sorted logic that is able to capture exactly the set of NP search problems over both the class of arithmetical structures and the class of finite structures. We believe that this multi-sorted logic can be used as a guideline to equip existing declarative languages (such as the system languages of IDP, Enfragmo, Gringo and Lparse) with arithmetical constructs so that the

expressiveness of these languages in the presence of arithmetic is still under control.

1.2 Combining languages and solvers

A more recent approach to address issues (†1) and (†2) is to combine solvers from different communities along with their respective languages. Using such combinations of languages, the non-specialist user can describe each of the constraints of a problem using the language that is most suited for modeling that constraint. Moreover, since there is no need to encode a construct using other constructs, a problem's structure will not be lost. Furthermore, since a problem's structure is not lost, the propagation/solving techniques that are developed to speed up the solving of specific constraints can still be used. One example in this direction is the effort of ASP community to devise efficient ways of embedding CP solvers into ASP solvers. Multi-context systems (MCS) are another example of an effort to combine knowledge bases under different semantics.

Such language combinations have traditionally been motivated in two ways: (a) To take advantage of different expressive capabilities provided by different constructs of the involved languages, and, (b) to reduce the conceptual complexity of a problem by breaking it into many (easier and less complex) sub-problems. Recently, the advent of online services and the growth of local knowledge bases to the global scale, has added a new reason to pursue the goal of a more seamless interaction of knowledge bases with different semantics.

The combination of all the above motivating factors has made a strong case for the development of a modular declarative framework that can work with modules in different languages. Thus, one of the most important requirements on such a framework should is its language-independence. This way, owners of all knowledge bases (declarative or non-declarative) can participate in the process of solving problems that would have been impossible to solve independently. Another important requirement on such a modular framework is to guarantee efficiency without requiring involved modules to expose all their information. Such a guarantee is specially important when knowledge bases contain data that are subject to privacy or confidentiality issues (e.g., when a knowledge base contains personal or business intelligence data).

The task of model expansion (that we introduced before) perfectly satisfies the requirements we need for designing our modular system framework: (1) It is language-independent because it views problems as sets of structure. (2) Also, model expansion only considers the input-output behavior of modules and not how a module is specified.

Therefore, in Chapter 4 of this thesis, we define a declarative modular framework that views

modules as model expansion tasks. We define the set of well-formed modular systems through several operations: (a) we compose modules together in both a serial and a parallel fashion, (b) we make new modules by hiding the auxiliary vocabulary of other modules, and, (c) we introduce new modules through a feedback from some outputs of a module to some of its inputs. We also define two complementary semantics for our well-formed modular systems: the model-theoretic semantics and the operational semantics. Both of these semantics respect our intuition that each module represents a model expansion task. However, the model-theoretical semantics views modules as classes of structures but the operational semantics views them as operators on structures. We show that these two complementary semantics are indeed equivalent, i.e., the operator associated with each modular system (under operational semantics) closely corresponds to the class of structures that is associated with that modular system (under model-theoretic semantics).

Moreover, we study the complexity and expressiveness of our modular system framework. We show several results that characterize the complexity and expressiveness of different operations on modular systems under different conditions. We also develop two algorithms to find solutions to modular systems. These algorithms both build on the ideas of lazy DPLL(T) model finding. We expand these ideas to the case of model expansion and show that the resulting algorithm works in accordance with the modern solvers in the fields of Integer Linear Programming, Satisfiability Modulo Theories, and combinations of Answer Set Programming and Constraint Programming. The difference between these two Algorithms is that the first one uses only our model-theoretic semantics for modular systems in order to find solutions while the second one uses both semantics and, thus, refines and speeds up the process of looking for solutions. As we will see, our second algorithm uses the operational view on modular systems to develop mechanisms that find approximate solutions to modular systems. This type of approximate solving is specific to our system and is not present in other general paradigms for combined solving.

Support for Modular Systems: One of the pitfalls of black-box modeling of problems in traditional declarative problem solving is the reduction in our ability of tracing back the undesired behavior of a system. Since our modular framework conforms to the black-box modeling principles, it is also prone to such a pitfall. Therefore, we need to develop a method that allows us to look deep into an unpredictable behavior when such a behavior happens. In practice, all such undesirable behaviors can be characterized by *some property that should be satisfied by all models but that is not*, i.e., the modular system has a model that falsifies our desired property.

In such cases, in order to debug a system, one needs to know why the desired property is not satisfied. To this end, in Chapter 5 of this thesis, we equip our modular framework with a new tool:

the *support functions*. Support functions are abstract functions that provide us with reasons about why certain beliefs are asserted by a modular system. They allow us to inspect the models of a modular system and choose only the "reasonable" models. Moreover, other "unreasonable" models of a modular system can now be traced back to their basic signs of unreasonableness.

Using support functions, we introduce a new semantics for modular systems that we call *supported semantics*. Similar to what was done for the previous semantics of modular systems, we study the complexity and expressiveness of modular systems under the new semantics. We show that the new extended modular system is indeed a proper extension of the previous semantics, i.e., it is more expressive than modular systems without support functions.

There are other important advantages to adding support functions to modular systems. For example, we show that, in the presence of support functions, modular systems naturally generalize both the equilibrium semantics and the grounded equilibrium semantics of multi-context systems [25]. Therefore, by means of support functions, one can make modular systems provably more expressive than multi-context systems under either of its semantics.

Supported Semantics for Multi-context Systems: Motivated by the far-reaching and positive consequences of equipping modular systems with support functions, and due to the close correspondence of modular systems to multi-context systems, in Chapter **??**, we also introduce the notion of support functions in multi-context systems and define a new semantics for multi-context systems that we call *supported equilibrium semantics*. We prove that the positive consequences of supported semantics in modular systems indeed carry over to multi-context systems as well. That is, we show that previously different semantics of normal and grounded equilibria can both be viewed as specializations of supported equilibrium semantics with different support function. In this sense, our new supported equilibrium semantics for multi-context systems can be viewed as a unifying semantics. That is, the more information the support functions provide about the inner workings of a context, the more refined the set of "chosen" models would be.

Moreover, we show that supported equilibrium semantics enables us to deeply inspect a contexts to see why particular belief states are or are not supported. This extended ability paves our way for providing a more insightful set of inconsistency explanations and/or diagnoses for faulty multi-context systems. Thus, under our new supported equilibrium semantics, a diagnosis can also reflect the possible changes to the knowledge base of different contexts that can restore the correctness of a faulty multi-context system.

Supported Semantics as a Rational Extension of Stable Model Semantics: One of the most important properties of supported semantics for modular systems is that it does not allow chains of

self-justification, i.e., beliefs in a supported model of a modular system are well-justified. Having well-justified beliefs is desired by many semantics and is also a basis for rationality of models. Among the semantics that only allow well-justified models, stable model semantics is perhaps the most famous one. However, surprisingly, this useful property is lost in both main-stream extensions of stable model semantics beyond the syntax of logic programs, i.e., equilibrium semantics and FLP semantics.

In order to restore the useful property of allowing only well-justified models, in Chapter 6, we extend the concept of support for beliefs to stable model semantics and define supported semantics for propositional logic. We show that, if restricted to logic programs, supported semantics coincides with stable model semantics. Moreover, we show that supported semantics share many properties such as minimality and well-justifiedness of beliefs with stable model semantics (unlike previous extensions of stable model semantics). Chapter 6 connects supported semantics to many other semantics for non-monotonic reasoning. We show that, although supported semantics is defined in terms of intuitionistic reasoning, for reasoning about supported semantics, full Kripke models are not required. In fact, we show that HT-models (the simplest form of non-classical Kripke models) are enough to fully characterize supported semantics. This fact leads us to several interesting results including that, despite guaranteeing the rationality of supported models, the complexity of reasoning about supported models does not increase, i.e., rationality guarantee comes for free.

Also, by defining stable models in terms of supported semantics, we are able to show (for the first time) an intrinsic connection between HT-models and stable model semantics. Previous connections between these two concepts have always been possibilistic in the sense that they show that a connection may exist but fail to show this connection is necessary. Our result, however, shows that stable model semantics is necessarily connected to HT-models because of the way stable models are defined.

Chapter 2

Background

2.1 Model Expansion

For each logical language, several tasks can be studied – satisfiability and model checking are among them. In this thesis, we are interested in search problems and, thus, we follow the authors of [149] who formalize combinatorial search problems as the task of *model expansion (MX)*, the logical task of expanding a given (mathematical) structure with new relations. Formally, the user axiomatizes the problem in some logic $\mathcal{L}$. 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/modelling language. It could be an extension of first-order logic such as FO(ID) [55], or an ASP language, or a modelling language from the CP community such as ESSENCE [76].

Recall that a vocabulary is a set of non-logical (predicate and function) symbols. An interpretation for a vocabulary is provided by a *structure*, which consists of a set, called the domain or universe and denoted by $dom(.)$, together with a collection of relations and (total) functions over the universe. A structure can be viewed as an *assignment* to the elements of the vocabulary. The *restriction* of a σ-structure $\mathcal{A}$ to vocabulary τ ($\tau \subseteq \sigma$), denoted by $\mathcal{A}|_\tau$, is the τ-structure $\mathcal{A}'$ with the same universe and the same interpretations as $\mathcal{A}$ (for vocabulary symbols in τ). An expansion of a structure $\mathcal{A}$ is a structure $\mathcal{B}$ with the same universe, and which has all the relations and functions of $\mathcal{A}$, plus some additional relations or functions. 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 $\varepsilon := vocab(\phi) \setminus \sigma$ the *expansion* vocabulary[1].

The complexity of model expansion task can be studied in two settings: the *combined complexity* where both both the formula and the instance structure are given and *data complexity* where the formula is fixed and the instance structure is given. In this thesis, since we are interested in search problems, we focus on *data complexity* of model expansion tasks.

Example 2.1 *The following formula ϕ of first order logic constitutes an MX specification for Graph 3-coloring:*

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

An 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 $\mathcal{G}$.

Given a specification, we can talk about a set of $\sigma \cup \varepsilon$-structures which satisfy the specification. Alternatively, we can also talk about a given set of $\sigma \cup \varepsilon$-structures as an MX-task, without mentioning a particular specification the structures satisfy.

[1] By ":=" we mean "is by definition" or "denotes".

2.2 Multi-contex Systems

This section briefly reviews multi-context systems. We use the exposition of [71] and [25] to describe multi-context systems. We use the two following notations in our exposition in this section and in other places throughout this thesis:

Notation 2.1 (Negating a Set) *Throughout this thesis, for a set of belief literals X, we use "***not** *X" to denote the set of negated literals for literals in X, i.e.,* **not** $X := \{\textbf{not}\ b \mid b \in X\}$.

Notation 2.2 (Un-pairing) *For a pair $P := (X, Y)$, we use $fst(P)$ to denote X and $snd(P)$ to denote Y.*

In multi-context systems (MCS: [25]), a *logic* L is defined by triple $L := (KB_L, BS_L, ACC_L)$, with KB_L being a set of knowledge bases (the syntactical fragment of logic L), BS_L being a set of belief sets (the semantical objects of logic L), and $ACC_L : KB_L \mapsto 2^{BS_L}$ being a mapping from knowledge bases to their acceptable belief sets (the semantics of logic L). A *multi-context system* $MCS := (C_1, \cdots, C_n)$ is a collection of contexts $C_i := (L_i, kb_i, br_i)$ with logic L_i, knowledge base $kb_i \in KB_{L_i}$ and bridge rules br_i. In MCSs, bridge rule $r \in br_i$ has the following form:

$$(i : s) \leftarrow (c_1 : p_1), \cdots, (c_j : p_j), \textbf{not}\ c_{j+1} : p_{j+1}, \cdots, \textbf{not}\ c_m : p_m. \tag{2.1}$$

where $hd(r) := s$; $body^+(r) := \{(c_k : p_k) \mid 1 \leq k \leq j\}$; $body^-(r) := \{(c_k : p_k) \mid j{+}1 \leq k \leq m\}$; and, $body(r) := body^+(r) \cup (\textbf{not}\ body^-(r))$.

A *belief state* $S := (S_1, ..., S_n)$ is a collection of belief sets, i.e., $S_i \in BS_{L_i}$. A bridge rule r of form (2.1) is *applicable* wrt. S, denoted by $S \models body(r)$, iff $p_l \in S_{c_l}$ for $1 \leq l \leq j$ and $p_l \notin S_{c_l}$ for $j < l \leq m$. We define $app_i(S) := \{hd(r) \mid r \in br_i \wedge S \models body(r)\}$ to obtain heads of all applicable bridge rules of context C_i. Belief state S is an *equilibrium* of *MCS* if, for all i, $S_i \in ACC_{L_i}(kb_i \cup app_i(S))$.

A logic L is *monotone* if (1) all $kb \in KB_L$ have a unique acceptable belief set S, i.e., $ACC_L(kb) = \{S\}$, and, (2) $S_1 \subseteq S_2$ whenever $kb_1 \subseteq kb_2$ (for $ACC_L(kb_1) = \{S_1\}$ and $ACC_L(kb_2) = \{S_2\}$). Also, a logic L is *reducible* if (1) subset $KB^*_L \subseteq KB_L$ exists s.t. $L^* := \langle KB^*_L, BS_L, ACC_L\rangle$ is monotone, and, (2) reduction function $red_L : KB_L \times BS_L \mapsto KB^*_L$ exists s.t. a. $red_L(kb, S) = kb$ if $kb \in KB^*_L$, b. $red_L(kb, S_2) \subseteq red_L(kb, S_1)$ whenever $S_1 \subseteq S_2$, and, c. $S \in ACC_L(k)$ iff $ACC_L(red_L(k, S)) = \{S\}$.A *context* $C := (L, kb, br)$ *is reducible* if its logic L is reducible and for all $H \subseteq \{hd(r) \mid r \in br\}$ we have $red_L(kb \cup H, S) = red_L(kb, S) \cup H$. MCS M is reducible if all of its contexts are reducible.

A reducible MCS $M := (C_1, \ldots, C_n)$ is *definite* if all bridge rules r of M are positive, i.e., $body^-(r) = \emptyset$, and, for all i, $kb_i \in KB^*_{L_i}$. Definite MCSs guarantee monotonic inference and, thus, always have a unique minimal equilibrium [25]. Moreover, for reducible MCS $M := (C_1, \cdots, C_n)$ and belief state S, reduction of M under S, denoted by M^S, is a definite MCS $M' := (C_1^S, \cdots, C_n^S)$ where $C_i^S := (L_i, red_i(kb_i, S_i), br_i^S)$ and $br_i^S := \{hd(r) \leftarrow body^+(r) \mid r \in br_i \text{ and } S \models body(r)\}$. Using these notations, a belief state S is the *grounded equilibrium* of reducible MCS M if S is the unique minimal equilibrium of the reduction of M under S, i.e., M^S. For non-reducible MCSs, grounded equilibria are not defined.

Note that there exists another equilibrium semantics for an extension of MCSs (known as managed MCSs [26]) that we do not consider here because they are meant to model change of knowledge in contexts.

Chapter 3

Built-in Arithmetic in Model Expansion

3.1 Introduction

Search Problems: Many computationally complex problems can be easily defined as problems in which we are given an input and we are interested in one of the outputs that satisfies all the constraints of the problem. Such problems are known as *search problems*. An example of a search problem is an airline's flight scheduling problem in which we are given a list of tickets an airline has issued, and a list of airplanes the airline has, and we are interested in an schedule for airplanes that takes all the passengers to their destinations with a minimal cost.

Declarative Search Languages: In the past decade, the knowledge representation (KR) community has developed many declarative modeling systems for solving exactly such hard search problems. These systems provide a *convenient* high-level language and an *efficient* solving mechanism (thanks to their fast underlying solvers). Examples of such declarative modeling systems include the Gringo system [79] for Answer Set Programming [32], ESSENCE framework [76] for constraint programming, and IDP system [212] for programming in an extension of first-order logic with inductive definitions (under well-founded semantics). These systems have been effectively applied to complex real-world problems. For example, BioASP [83] is a library of answer set programs for solving complex biological problems.

Expressiveness and Naturalness of Declarative Languages: Two principles govern the usability of a declarative language: (1) its *expressiveness*, and, (2) its *naturalness*. The expressiveness of a language talks about the language's ability to axiomatize a class of problems and the naturalness of a language talks about the simplicity of specifying a class of problems in a language. Characterizing

expressiveness of a language addresses two questions simultaneously: (a) lower bounding a language's expressiveness give us *universality* results, and (b) upper bounding its expressiveness give us *complexity* results about the language. For a language L and a class C of problems, a universality result says that L can express all problems in C and a complexity result says that the computational power needed to solve problems in L is less than the computational power needed for solving problems in C (because all problems in L are also in C). Therefore, to a programmer, universality results guarantee that L is *useful* (because it can be used to solve problems in C) and complexity results guarantee that L is *feasible* (because they upper bound the computational resources needed to solve problems in L).

Complexity versus Expressiveness: The computational complexity of a language is the lowest complexity class that contains all the problems expressible in that language. Therefore, the higher a language's expressiveness, the higher its computational complexity. However, vice versa is most of the time not true, i.e., for a language L and its complexity class C, problem $P \in C$ exists such that P is not expressible in L. When this is not the case, i.e., when C is exactly the set of problems expressible in L, we say that L *captures* complexity class C. Capturing is an important computational property because it happens when L's computational power coincides with L's expressive power.

Arithmetic in Declarative Languages: Arithmetic, due to its importance in KR, is among the few features that are present in the syntax of all practical declarative modeling systems. However, since unlimited arithmetical syntax makes a language undecidable, each declarative language limits the syntax of its arithmetic in its own way. As we will show in this chapter, such arbitrary syntactical limitations has led to a huge disparity between the expressiveness and the complexity of existing declarative languages. *The main motivation and goal of this chapter is to show that the regularity between the computational power and the expressive power of declarative KR languages can be restored even in the presence of arithmetic.*

3.1.1 Our Goals

Our long-term research goal is to develop mathematical foundations of adding external functions to declarative languages for representing and solving search problems. We aim at developing a framework which (a) represents knowledge in its most *natural* form; and, (b) provides easy and sufficient means for *controlling the expressiveness* of a language. Naturalness is important for usability of a framework and controlled expressiveness is important for its feasibility (by upper bounding the language's complexity).

Remark 3.1 *We hope that our research on controlled expressiveness leads to developing* complexity-aware declarative languages*, i.e., practical declarative languages whose different fragments capture different complexity classes. This way, novice users who do not understand complexity classes can also be advised to use the more restrictive fragments of a language for writing their programs. Hence, users can achieve better solving performance without having to understand computational complexity classes.*

In this chapter, *our goal is to apply the concept controlled expressiveness to arithmetical search problems*. Towards this goal, we follow two sub-goals:

1. We want to analyze the expressiveness of arithmetical in existing declarative languages for knowledge presentation and discover both their comparative strengths and their comparative shortcomings.
2. We want to design a logic that (a) has built-in arithmetic, (b) has controlled expressiveness, and, (c) is natural. For example, this language should enable its users to quantify over the infinite domain of integers without being undecidable.

3.1.2 Previous Closely Related Work

As stated previously, this chapter aims to develop a formalism (with controlled expressiveness) for specifying arithmetical search problems. A partial solution, inspired by a previous proposal [100], was given in [189]. There, *embedded model expansion* was introduced to formalize arithmetical search problems. There, arithmetical structures included both built-in operations $\times$, $+$, $<$, etc, and built-in aggregate functions (e.g., min and sum). The authors needed a method for handling operations with outputs outside of the input domain, as is common in practical languages. They also desired universal quantification over integers since it is convenient and is used in practice. Access to the arithmetical structure through weight terms as in [100] was not sufficient. Thus, they defined two new logics, GGF$_k$ and DGGF$_k$. The former is an extension of the k-guarded fragment of FO (or FO(ID)), in which instance predicates are used as guards of quantifiers and expansion predicates (here, GG stands for double-guarded, not for [100]). DGGF$_k$, is an extension of GGF$_k$ in which definable guards are allowed, provided they are polysize in the domain size. The extension allows for quantifying over variables whose values fall outside of the input domain. It was proven that, under a small-cost condition[1], NP is captured for both fragments. That is, (a) for every small-cost

[1]Small-cost condition says that numerical input values should be bounded by a function that grows at most exponentially in domain size.

problem in NP (represented by a class of logical structures) there is a specification in these logics such that an instance (a structure) is in the class if and only if there is an expansion of that structure that satisfies the specification; and (b) for every specification in these logics, the task of model expansion is in NP.

Although these two fragments provide natural axiomatizations that do not involve binary encodings for some problems, they are limited in two important ways:

1. **Poly-size guards** are too limiting. Suppose we want to output the total weight of all items in a $Knapsack$ (an expansion predicate). A natural axiom would be:

$$\exists x\ (G(x) \wedge Output(x) \wedge x = \Sigma_y(\ Weight(y)\ :\ Knapsack(y)).$$

   We cannot use a polysize guard G: there are up to 2^n distinct sums, where n is the size of the domain.

2. **Small cost condition** cannot be satisfied in natural axiomatizations of some common problems in NP such as integer factorization or a quadratic programming problem:

$$\text{given } m, a, c \text{ find } x \text{ such that } x^2 = a\ (\mathbf{mod}\ m) \wedge x > c.$$

   The values of the given integers, m and a, are, in general, unlimited in the size of the input domain, which is 3.

3.1.3 Contributions

In this chapter, we both study the expressiveness of practical KR languages and propose a new ideal language (w.r.t. arithmetic). The three existing KR languages that we study are the Gringo system and the Lparse system (for answer set programming), and the IDP system (for programming in first-order logic extended with inductive definitions). However, the expressiveness and inexpressiveness results we obtain in this chapter are applicable to many other existing KR systems, e.g., the NP-SPEC [28] modeling system.

Studying Expressiveness of Built-in Arithmetic in Existing KR Languages

To the best of our knowledge, we are the first to formally study the expressiveness of KR languages in the presence of arithmetic[2]. We prove lower and upper bounds on the expressiveness of built-in

[2]A possible exception that worth noting is [151] in which the authors study the expressiveness of ESSENCE, a constraint programming language.

arithmetic in declarative KR languages. We also prove inexpressiveness results that hold regardless of the presence of fixpoint constructions in these languages (all these languages include some sort of construction). The following summarizes our contributions in this regard.

1. Lower Bounding the Expressiveness of Existing KR Languages: We show that the built-in arithmetic of the three languages we study, i.e., Gringo, Lparse and IDP is powerful enough to express all the small-cost arithmetical structures that can be recognized in NP. We obtain this result by showing all specifications in logic GGF_k can be easily translated to specifications in these languages.

2. Upper Bounding the Expressiveness of Existing KR Languages: We introduce three categories of logic: *domain-restricted* logics, *polytime domain-restricted* logics and *loosely domain-restricted* logics. We show that each of the three languages of Gringo, Lparse and IDP falls into one of these categories.

3. Concrete Inexpressiveness Results for the Existing KR Languages: We use the upper bounds we obtained on languages of Gringo, Lparse and IDP to show that, under complexity-theoretical conditions, none of these languages can express some common computational problems such as integer factorization.

4. Non-conditional Limit on the Size of Numbers in ASP Languages: We show that, for the languages of Gringo and Lparse, the number of bits needed to represent any output number grows linearly in the number of bits used by the maximum input value. We use this fact to ptove non-conditional inexpressibility results for these languages.

Logic PBINT

We introduce new logic PBINT that captures exactly the arithmetical problems that are NP-recognizable. Our contributions in this regard are as follows:

1. Eliminating Small-cost Condition: Our new logic PBINT eliminates the small cost condition $\text{GGF}_k(\varepsilon)$ had imposed on the capturing result of [189] and unconditionally captures NP for all problems involving arithmetic. Thus, PBINT is an ideal specification/modelling language with respect to arithmetic.

2. Different Background Structure: To obtain our unconditional capturing result, PBINT uses a new background structure than that of [189], namely the structure containing at least $(\mathbb{N}; 0, 1, +, \times, <, ||\ ||)$, among other polytime relations. The key difference between the new background structure and the previous one is the operator $||\ ||$ that calculates the binary encoding size of a number.

3. Natural Representation of Arithmetical Problems: PBINT guarantees that all arithmetical problems in NP have a natural axiomatization, i.e., an axiomatization that uses numbers and numerical operations without having to encode numbers and numerical operations using abstract domain elements.

4. New Types of Guards: Unlike in [189], PBINT allows exponential size guards while carefully restricting their use. Moreover, PBINT also allows guards with size polynomial in the size of binary encoding of the structure. These two new types of guards contribute to our final unconditional capturing result for PBINT.

5. Polytime Grounding: Using the new types of guards in a careful way, PBINT guarantees that all its specifications are recognizable in NP and, thus, have polytime grounding (in the size of binary encoding of the structure).

6. Capturing NP for Arithmetical Problems: Finally, PBINT captures NP for arithmetical problems, i.e., (1) nothing outside NP is expressible in PBINT and (2) for every problem in NP involving arithmetic (and numbers), there is an PBINT specification ϕ of that problem which works with numbers as numbers (instead of encoding them using abstract domain elements).

Note that a common misconception is that proving hardness of a language $\mathcal{L}$ for a complexity class C is assumed to be equivalent to proving that $\mathcal{L}$ can express all problems in C. This assumption is false and its falsity, for example, is shown in Section 3.4 where we show that some languages can express many NP-hard problems but not the integer factorization problem (which is in NP). Of course, since integer factorization is in NP and since these languages can express some NP-complete problems, the integer factorization problems can be *reduced* to a problem expressible in these languages but one should note the difference between reducibility and expressibility. Hardness properties are about computational power and not expressive power.

PBINT-Spec

Based on PBINT, we introduce an idealized declarative language that we call PBINT-Spec. PBINT-Spec has all the properties of PBINT as listed above plus many of the common features of existing declarative languages (such as being multi-sorted). In Section 3.6, we show how some of our motivating examples from Section 3.2 are specified in PBINT-Spec.

The rest of this chapter is organized as follows. Section 3.2 provides some motivation and guidelines for the rest of this chapter. It introduces many examples of different arithmetical search problems and gives at least one ideal axiomatization for each of these problems. Section 3.2 can be safely skipped to study the main content and results of this chapter. Section 6.2 provides the

background needed in this chapter. Section 3.4 gives the first part of this chapter's contribution by formally investigating the expressive power of some of existing knowledge representation languages in the presence of their arithmetical constructs. We show both positive and negative expressiveness results in this regard. In Section 3.5, we introduce a novel logical fragment, called PBINT, which captures exactly those search problems involving arithmetic that are decidable in NP. Section 3.6 gives some sample PBINT specifications of problems involving arithmetic to further demonstrate the naturalness of PBINT specifications in practice. In the end, Section 3.7 reviews the literature surrounding the concept of arithmetic in natural modeling languages and contrasts this work against the previous works in this domain.

3.2 Motivating Examples

Throughout this chapter, we make abundant reference to *"naturalness"* of specifications. However, naturalness is a vague qualitative property that cannot be formally investigated. Although we formalize our understanding of naturalness later on, this section motivates our notion of naturalness by means of several examples of search problems that deeply rely on arithmetic. For each such example, we provide at least one "ideal" specification to help the reader have an intuition of how natural specifications look like and how they differ from unnatural specifications. Note that our ideal specifications may use features that are currently non-existent in KR languages (i.e., ideal specifications are written in idealized versions of KR languages). We have also chosen our examples such that each of them represent a category of arithmetical search problems. Thus, expressiveness of one of our examples in a KR language indicates the capability of that language to also express all other problems in the category that our example belongs to. Thus, *expressiveness of different KR languages can be informally compared with respect to what subset of our examples they can naturally express*.

Example 3.1 (Blocked N-Queens) *In the blocked N-queens problem you are given two unary relations Row and $Column$ and a binary relation $Blocked$. The two unary relations of Row and $Column$ each contain all the numbers 1 to N (for some N) and, together, they define an N by N chess board. The relation $Blocked$ contains a subset of positions on the board where queens are forbidden to be placed. You are asked to find (if possible) a placement for N queens on the non-blocked squares such that no two of them can capture each other. A solution to this problem puts the queens on distinct rows, distinct columns and distinct diagonals of the board. While asserting*

that queens should be placed on distinct rows and distinct columns can be easily described using non-arithmetical constructs, the assertion that diagonals should also be distinct is best described in the presence of arithmetic. For example, an ideal ASP specification for this problem looks as below:

$$1\{Queen(X,Y) : Row(X)\}1 \leftarrow Column(Y).$$
$$1\{Queen(X,Y) : Column(Y)\}1 \leftarrow Row(X).$$
$$\bot \leftarrow Queen(X,Y), Blocked(X,Y).$$
$$\bot \leftarrow Queen(X_1,Y_1), Queen(X_2,Y_2), |X_1 - X_2| = |Y_1 - Y_2|.$$

where "$-$" and "$|.|$", as is common, denote the subtraction (binary) operation and the absolute value (unary) operation. Also, $a\{.\}b$ is an aggregate formula that restricts the number of true atoms in the specified set to a number between a and b. A first order axiomatization of the blocked N-Queens problem ideally looks like as the following:

$$\forall r\ (\exists c\ (Queen(r,c))),$$
$$\forall c\ (\exists r\ (Queen(r,c))),$$
$$\forall r \forall c\ (Blocked(r,c) \supset \neg Queen(r,c)),$$
$$\forall r \forall c_1 \forall c_2\ (Queen(r,c_1) \wedge Queen(r,c_2) \supset c_1 = c_2),$$
$$\forall c \forall r_1 \forall r_2\ (Queen(r_1,c) \wedge Queen(r_2,c) \supset r_1 = r_2),$$
$$\forall r_1 \forall c_1 \forall r_2 \forall c_2\ (Queen(r_!,c_1) \wedge Queen(r_2,c_2) \wedge r_1 > r_2 \supset |r_1 - r_2| \neq |c_1 - c_2|).$$

The blocked N-Queens problem is an NP-complete problem that showcases one of the many interesting puzzles that, although easy to define, are computationally intractable. For a specification language, being able to axiomatize the blocked N-Queens problem shows that the language supports basic arithmetical constructs.

Example 3.2 (Knapsack) *In the Knapsack problem, you are given a set I of items and their associated weights $W : I \mapsto \mathbb{N}$ and values $V : I \mapsto \mathbb{N}$. You have a knapsack of a given limited weight capacity W_k and you are asked to find a subset C of items which can be packed into your knapsack and whose value is more than a given number V_k. An ideal first-order specification of the knapsack problem uses sum aggregates as follows:*

$$\forall x(C(x) \supset Item(x)),$$
$$\Sigma_x(W(x) : C(x)) \leq W_k,$$
$$\Sigma_x(V(x) : C(x)) \geq V_k.$$

The knapsack problem is an NP-complete problem that can be solved using dynamic programming and with a runtime linear in the maximum value of the input (which is exponential in the size of input as the input is represented in binary). The knapsack problem showcases a range of problems that are called pseudo-polynomial, i.e., polynomially solvable in the value of the input (not the size). For a specification language to be able to axiomatize the Knapsack problem naturally, it should (1) support aggregate constructs, and, (2) be able to deal with pseudo-polynomial problems.

Example 3.3 (Traveling Salesman Problem) *In the Travelling Salesman Problem (TSP), you are given a weighted graph G with vertices V and edges E. The weight of each edge $e \in E$ is given by $W(e)$ (where $W : E \mapsto \mathbb{N}$). You are also given a maximum distance $d \in \mathbb{N}$ and asked to find a simple cycle C in G whose weight does not exceed d. Anser set programs can conveniently express all the non-arithmetic constraints (i.e., all constraints except the weight constrant). Extensions of the answer set programming languages (e.g., [32]) can represent the summation construct as well. Therefore, an ideal ASP specification for this problem looks like below:*

$$
\begin{aligned}
&1\{C(X,Y) : E(X,Y)\}1 \leftarrow V(X).\\
&R(X,Y) \leftarrow C(X,Y).\\
&R(X,Z) \leftarrow R(X,Y), R(Y,Z).\\
&\bot \leftarrow V(X), V(Y), \mathbf{not}\ R(X,Y).\\
&\bot \leftarrow d < \mathrm{SUM}\{W(X,Y) : C(X,Y)\}.
\end{aligned}
$$

In the above axiomatization, the first line says that each vertex should have exactly one outgoing edge in the cycle. The next two lines define a reachability relation using the edges included in the cycle and the line after that asserts that every two vertex should be reachable in this way. The last line restricts the set of accepted cycles to those whose total cost does not exceed d. A natural axiomatization of this problem in ID-logic [55] would also be similar:

$$
\begin{aligned}
&\forall x \forall y\ (E(x,y) \supset C(x,y)).\\
&\forall x \forall y_1 \forall y_2\ (C(x,y_1) \wedge C(x,y_2) \supset y_1 = y_2).\\
&\{\forall x \forall y\ (R(x,y) \leftarrow C(x,y) \vee \exists z\ (R(x,z) \wedge R(z,y))).\}.\\
&\forall x \forall y\ (R(x,y)).\\
&d \geq \Sigma_{x,y}(W(x,y) : C(x,y)).
\end{aligned}
$$

TSP is also an NP-complete problem. However, this problem is harder than the two previous ones as it is also inapproximable up to any polynomial factor (unless P=NP). More importantly, TSP is the basis for a wide spectrum of industrial problems known as vehicle scheduling. There has

been numerous efforts to solve big instances of the TSP problem [12]. Usually, any method applied for solving this problem can be extended to solve industrial vehicle scheduling problems. Therefore, for a specification language which aims at solving hard industrial problems such as different vehicle routing problems, it should first be able to naturally represent the TSP problem.

Example 3.4 (Integer Factorization) *Here, you are given two numbers n and c and you are asked to find a non-trivial positive divisor k of n which is smaller than c. The ideal first-order specification of this problem is as follows:*

$$\exists k' \, (k \times k' = n) \wedge k \leq c \wedge k > 1 \wedge k < n.$$

The integer factorization problem is believed neither to be NP-complete nor to be polynomial-time solvable. This problem is the cornerstone of many cryptographic systems. Thus, there has been wide interest in studying methods of solving integer factorization. One of the interesting differences between this example and the previous motivating examples is that each instance of the problem is described with only a few numbers. This is unlike previous examples where, if the number of given items, cities, etc. were limited, the instance became trivial. So, this example shows that even a few numbers (two in this example) can define a complex mathematical problem.

Example 3.5 (Quadratic Residues) *In the quadratic residues problem, you are given three numbers n, a and c and you are asked to find the modulo-n square root of a which is smaller than c, i.e., a number s such that $s^2 \equiv a \pmod{n}$ and $s < c$. An ideal first-order specification for this problem is:*

$$0 \leq s \wedge s \leq c \wedge s < n \wedge \exists k \, (0 \leq k \wedge k < n \wedge s \times s = k \times n + a).$$

The quadratic residue problem is an important example of an arithmetical problem that, despite the fact that it uses only a few numbers, is still NP-complete. Together with the problem of integer factorization, these examples demonstrate that in order for a fragment of a specification language to capture all arithmetical NP search problems, it should be able to represent a wide variety of complex problems such as those with only a few numbers involved.

Example 3.6 (Prime Factor Permutation) *Another interesting problem is the prime factor permutation problem. It is not defined in the literature elsewhere (as far as the authors know). Here, you are given a number n and asked if there exists a different number m which is obtained from n by*

permuting the prime bases in its prime factorization. For example, when n is 20, its prime factorization is $2^2 \times 5^1$, and by permuting its prime bases, we can get $5^2 \times 2^1 = 50$. However, for $n = 100$, its prime factorization is $2^2 \times 5^2$ and permuting the prime bases does not yield a new number.

This problem is easier than factorization (hence, it is in NP) but it is interesting because the answer to this problem may need quadratic as many bits to represent as does the input. For instance, consider a number n of the form $2^k \times p$ where p is a prime number that needs k bits to represent. Thus, we need $2k$ bits to represent n, i.e., k bits for each one of the two numbers 2^k and p. However, the only solution m for such n is $m = 2 \times p^k$ which needs about $k^2 + 1$ bits to represent. As we will see later in this chapter, many systems cannot represent such a problem for exactly this reason of needing more than a linear number of bits.

An ideal specification to this problem looks as follows. Given number n, find m such that:

$$\exists p, q, k, k' \left(\begin{array}{c} k' \neq k \wedge A(n,p,k) \wedge A(n,q,k') \wedge A(m,p,k') \\ \wedge A(m,q,k) \wedge \frac{n}{p^k \times q^{k'}} = \frac{m}{p^{k'} \times q^k} \end{array} \right),$$

where $A(n,p,k)$ denotes that prime number p appears with exponent k in the prime factorization of n. Also, $P(p)$ indicates that p is a prime number. These two are defined as follows:

$$A(n,p,k) := P(p) \wedge k > 0 \wedge p^k \mid n \wedge p^{k+1} \nmid n,$$
$$P(p) := \neg \exists m (p > m \wedge m > 1 \wedge m | p).$$

Here, $m|p$ means that p divides m and $m \nmid p$ is the negation of that property.

As we see from these example, a natural way to deal with arithmetic in specifications languages of search problems as the ones above would:

- be able to use an infinite structure of arithmetic,
- not be required to axiomatize the built-in operations,
- not be forced to use modulo n operations, where n is the size of a finite domain,
- be able to quantify over numbers in the infinite domain.

We want to develop a formalism where there are as few syntactic limitations as possible and all of the requirements above are satisfied. At the same time, we want to be able to develop it so that we are able to control its expressive power. We undertake this task for an extension of first-order logic. We expect the same ideas to be applicable for Answer Set Programming, but we leave this extension to a later work.

3.3 Background

Throughout this chapter, we use $\exists\bar{x}$ to denote $\exists x_1 \ldots \exists x_n$ and $\forall\bar{x}$ to denote $\forall x_1 \ldots \forall x_n$.

Embedded MX

Embedded finite model theory (see [132, 133]), the study of finite structures with domain drawn from some infinite structure, was introduced to study databases containing numbers and numerical constraints. Rather than a database being a finite structure, we take it to be a set of finite relations over an infinite domain.

Definition 3.1 *A structure $\mathcal{A}$ is* embedded *in an infinite* background *(or* secondary*) structure $\mathcal{M} = (U; \bar{M})$ if it is a structure $\mathcal{A} = (U; \bar{R})$ with a finite set $\bar{R}$ of finite relations and functions, where $\bar{M} \cap \bar{R} = \emptyset$. The set of elements of U that occur in some relation of $\mathcal{A}$ is the* active domain of $\mathcal{A}$*, denoted $adom_{\mathcal{A}}$.*

In database research, embedded structures are used with logics for expressing queries. Here, we use them similarly, with logics for MX specifications. Throughout, we use the following conventions: σ denotes the vocabulary of the embedded structure $\mathcal{A} = (U; \bar{R})$, which is the instance structure; ν denotes the vocabulary of an infinite background structure $\mathcal{M} = (U; \bar{M})$; ε is an expansion vocabulary; A formula ϕ over $\sigma \cup \nu \cup \varepsilon$ constitutes an MX specification. The model expansion task remains the same: expand a (now embedded) σ-structure to satisfy ϕ.

GGF$_k$ Logical Fragment

The authors of [189] use a guarded logic in an embedded setting, which allows them to quantify over elements of the background structure (unlike, e.g. [100]). To do so, they use an adaptation of the guarded fragment GF$_k$ of FO [92]. In formulas of GF$_k$, a conjunction of up to k atoms acts as a *guard* for each quantified variable.

Definition 3.2 *The k-*guarded fragment *GF$_k$ of FO (with respect to σ) is the smallest set of formulas that:*

1. *contains all atomic formulas;*
2. *is closed under Boolean operations;*
3. *contains $\exists\bar{x}\,(G_1 \wedge \ldots \wedge G_m \wedge \phi)$, provided the G_i are atomic formulas of σ, $m \leq k$, $\phi \in GF_k$, and each free variable of ϕ appears in some G_i.*

4. *contains* $\forall\bar{x}\ (G_1 \wedge \ldots \wedge G_m \supset \phi)$ *provided the* G_i *are atomic formulas of* σ, $m \leq k$, $\phi \in GF_k$, *and each free variable of* ϕ *appears in some* G_i.
   For a formula $\psi := \exists\bar{x}\ (G_1 \wedge \ldots \wedge G_m \wedge \phi)$, *conjunction* $G_1 \wedge \ldots \wedge G_m$ *is called the* existential guard *of the tuple of quantifiers* $\exists\bar{x}$; universal guard *is defined similarly.*

Example 3.7 Let ε be $\{E_1, E_2\}$. The following formula is not guarded: $\forall x \forall y\ (E_1(x, y) \supset E_2(x, y))$. It is guarded when E_1 is replaced by P which is not in ε. The following formula is the standard encoding of the temporal formula $Until(P_1, P_2)$: $\exists v_2\ (R(v_1, v_2) \wedge P_2(v_2) \wedge \forall v_3\ (R(v_1, v_3) \wedge R(v_3, v_2) \supset P_1(v_3)))$. The formula is 2-guarded, i.e., is in GF_2, but it is not 1-guarded.

The guards of GF_k are used to restrict the range of quantifiers. They also use "upper guard" axioms, which restrict the elements in expansion relations to those occurring in the interpretation of guard atoms. To formalize this, they introduce the following restriction of FO, denoted $\text{GGF}_k(\varepsilon)$.

Definition 3.3 *The double-guarded fragment* $\text{GGF}_k(\varepsilon)$ *of FO, for a given vocabulary* ε, *is the set of formulas of the form* $\phi \wedge \psi$, *with* $\varepsilon \subset vocab(\phi \wedge \psi)$, *where* ϕ *is a formula of* GF_k, *and* ψ *is a conjunction of* upper guard axioms, *one for each symbol of* ε *occurring in* ψ, *of the form* $\forall\bar{x}\ (E(\bar{x}) \supset G_1(\bar{x}_1) \wedge \cdots \wedge G_m(\bar{x}_m))$, *where* $m \leq k$, *and the union of free variables in the* G_i *is precisely* $\bar{x}$.

Guards of GF_k, that restrict quantifier ranges, are *lower guards*, and guards of Def. 3.3 are *upper guards*. In GGF_k, all upper and lower guards are from the instance vocabulary σ, so ranges of quantifiers and expansion predicates are explicitly limited to $adom_{\mathcal{A}}$.

To finish definition of the logic, they define well-formed terms which depends on the vocabulary of the background structure. The authors of [189] use *arithmetical structures*, same as [100]. They also introduce a fragment of arithmetical structures known as *small cost arithmetical structures*. They prove that the model expansion task for GGF_k captures NP for small cost arithmetical structures[3]. This chapter continues on their path and proves that the property of capturing NP over small cost arithmetical structures can be extended to several practical KR languages.

[3]We use the definition of arithmetical structures and small cost arithmetical structures in this chapter. So, for presentation reasons, these definitions are moved from background section to where they are used in Section 3.4.1.

3.4 Capturing and Non-Expressibility Results for Practical KR Languages

This section studies the expressiveness of built-in arithmetic in existing KR languages. But, in order to formally investigate the expressibility of KR languages, we first need to define what we mean by a specific language. We have based our investigations on system manuals published for these languages. However, since such manuals often neglect the details, we have also had to do several experiments in order to understand if particular types of specifications are allowed or not. In the end, we believe that our results are true of the specific languages that we will talk about.

As the case with all practical languages, they evolve over time and their syntax changes from one version to another. So, what is not allowed in current version, may be allowed in the next version or vice versa. Therefore, we specify the exact manual and version of these languages as below:

1. IDP version 1.4.4 and the accompanying manual published in August 2009 [210].
2. Gringo version 2.0.5 and the accompanying manual published in November 2008 [79].
3. Lparse version 1.1.2 and the accompanying Lparse 1.0 user's manual [182].

In this section, we will frequently refer to a common concept in many practical declarative languages that we call *compact domain representation*. While there is no universally accepted definition for this concept that is shared between all practical declarative languages, the concept still shows itself in similar forms and with similar intended meaning in different languages. In general, by compact domain representation, we refer to the operator that says something is true about a range of (integral) numbers by just giving the lower bound and the upper bound of the range. For example, in the system language of IDP [210], one can define a type using the expression $[1..n]$ with n being a number that would be later interpreted in the input. Similarly, in the system language of Gringo [79], one can have a rule of form "$R(1..X) \leftarrow I(X)$" that, informally, says $R(Y)$ is true if Y is an integer less than some integer in I. Since, in both these languages and in many other languages, such definitions work as a compact way to represent a huge active domain, we have given them the name of *compact domain representations*. We also want to mention that these types of definitions in practical languages are usually considered just a shorthand for the whole set and are treated exactly so, i.e., in the very beginning of the solving process, formulas/types with ranges are replaced by formulas/types about explicitly the elements in that range. Nevertheless, since we are treating expressiveness of languages in this chapter and since compact domain representations are able to increase expressiveness, we will also consider compact domain representations here.

3.4.1 Capturing Results

Definition 3.4 *An* Arithmetical structure *is a structure $\mathcal{N}$ containing at least*

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

with domain $\mathbb{N}$, the natural numbers, and where min, max, Σ and Π are multi-set operations[4] *and $\chi[\phi](\bar{x})$ is the characteristic function. Other functions, predicates, and multi-set operations may be included, provided every function and relation of $\mathcal{N}$ is polytime computable.*

Definition 3.5 *For an embedded arithmetical structure $\mathcal{D}$, define $cost(\mathcal{D})$, the cost of $\mathcal{D}$, to be $\lceil \log_2(l+1) \rceil$, where l is the largest number in $adom_{\mathcal{A}}$.*

Definition 3.6 (Small cost structures) *A class $\mathcal{K}$ of embedded arithmetical structures has small cost if there is some $k \in \mathbb{N}$ such that $cost(\mathcal{D}) \leq |adom_{\mathcal{D}}|^k$, for every $\mathcal{D} \in \mathcal{K}$.*

Next, we are going to give a theorem that relates the expressive power of ASP over arithmetical structures to the expressive power of GGF$_k$ fragment of logic. However, ASP programs are traditionally defined over relational structures and, thus, we have to extend this notion to arbitrary structures. In order to do this, we define *λ-restricted ASP Programs*.

Definition 3.7 (λ-restricted ASP Programs) *Let $B(v, r)$ denote the set of predicate symbols that appear in the body of rule r with variable v as a term in it, i.e., the term consists of only variable v. Also, let $V(r)$ denote all the free variables in r and $R_\Pi(p)$ denote all rules in ASP program Π with predicate symbol p in their head. Also, let $M(r)$ denote all multi-set terms t of r and $V_m(m)$ denote all variables that are quantified by multiset operation m and $B_m(v, m)$ denote all predicate symbols that appear as a positive atom in multi-set operation m and with variable v as one of their arguments. We say that an ASP program Π is* λ-restricted *if there is a function λ from predicate symbols of Π to natural numbers such that for all predicate symbols $\mathcal{S}$ in vocaulary of Π:*

$$\max\{\min\{\lambda(\mathcal{T}) \mid \mathcal{T} \in B(v, r)\} \mid v \in V(r),\ r \in R_\Pi(\mathcal{S})\} < \lambda(\mathcal{S}), \text{ and,}$$
$$\max\{\min\{\lambda(\mathcal{T}) \mid \mathcal{T} \in B_m(v, m)\} \mid v \in V_m(m),\ m \in M(r),\ r \in R_\Pi(\mathcal{S})\} < \lambda(\mathcal{S})$$

Note that, although the motivation for Definition 3.7 comes from Gringo, it is in fact different from both λ-restrictedness in [86] and level-restrictedness in [79]. To see why Definition 3.7 is

[4] *Multi-sets are generalizations of sets that allow multiple occurrence of elements.*

different from λ-restrictedness defined in [86], look at the following example which is not accepted by Definition 3.7 but is λ-restricted due to [86]:

$$q(0).$$
$$p(x) \leftarrow q(0 \times x).$$

Also, to see why Definition 3.7 is different from level-restrictedness defined in [79], observe that assignment operator is not present in Definition 3.7. In fact, Definition 3.7 accepts a subclass of ASP programs that are characterized by level-restrictedness property which is, in turn, generalized to safety property in newer versions of Gringo software [80]. The reason we use this limited class of ASP programs is to define a subclass of ASP programs that remains inside NP in the presence of arithmetic constructs. This work, for example, could be used in practical ASP solvers as a switch to either limit users to NP and give them a performance guarantee in return or to give them access to full ASP language but without such performance guarantees.

Now, we can use the notion of λ-restricted ASP programs and define admissible ASP programs over arithmetical structures to be those *ASP programs over such background structures that are also λ-restricted.*

Theorem 3.1 *Let $\mathcal{K}$ be a class of small-cost embedded arithmetical structures over vocabulary $\sigma \cup \epsilon \cup \nu$. Then the following are equivalent:*

1. *$\mathcal{K} \in$ NP;*
2. *There is a first order formula ϕ of GGF$_k(\varepsilon)$ such that $\mathcal{D} \in \mathcal{K}$ if and only if there is an expansion $\mathcal{D}'$ of $\mathcal{D}$ to ε so that $\mathcal{D}' \models \phi$;*
3. *There is a safe ASP program P with instance vocabulary σ such that $\mathcal{D} \in \mathcal{K}$ iff there is an expansion $\mathcal{D}'$ of $\mathcal{D}$ so that $\mathcal{D}'$ is a stable model for P.*

Proof: $(1) \Rightarrow (2)$ is shown in [189].
$(2) \Rightarrow (3)$ is shown by Lloyd-Topor transformation. We first push all negations inside and then introduce new relation symbols for negated expansion predicates. For example, for expansion predicate $E(\bar{x})$, new relation symbol $E'(\bar{x})$ is introduced and two following sentences are added to the ASP program:

$$E(\bar{x}) \leftarrow \text{ not } E'(\bar{x}).$$
$$E'(\bar{x}) \leftarrow \text{ not } E(\bar{x}).$$

Also, new relation symbols are introduced for each subformula and Lloyd-Topor transformation is used to relate these subformulas together in an appropriate way. However, the resulting ASP program is not still safe.

In order to convert this ASP program into a safe ASP program, we have to incorporate information from lower and upper guards in the GGF_k specification. Such guards define a permissible domain for each subformula. Therefore, we introduce new domain predicates based on this information. These predicates can be defined using a completely positive and non-recursive ASP program. So, all rules generated above can be modified so as their variables are bounded by these new domain predicates.

Note that, here, we do not claim that a first order formula ϕ can be translated into a safe ASP program via Lloyd-Topor transformation. What we claim (and what is needed here for the proof) is that, informally, for a specification ϕ in $\text{GGF}_k(\varepsilon)$, there is a safe ASP program P such that for all instance structures $\mathcal{D}$, there is a certificate for $\mathcal{D}$ in the $\text{GGF}_k(\varepsilon)$ sense iff there is a certificate for $\mathcal{D}$ in the ASP sense.

The key here is that the transformation does not have to preserve the entire models of the GGF_k formula. What is needed to be presereved is the existence of an expansion (and not the equivalence of the set of expansions) for a given instance structure.

Also, the expansion vocabulary of the ASP program does not have to be the same as ε.

$(3) \Rightarrow (1)$ is shown by giving a machine in NP that first guesses a stable model and then checks its stability in polytime. The existence of such a guessing procedure is guaranteed by the safety property of the ASP program. ∎

Corollary 3.1 *ASP language of Lparse and Gringo captures small cost arithmetical NP problems.*

Corollary 3.2 *The IDP language captures the small cost arithmetical NP problems.*

Proof: By Theorem 3.1, GGF_k covers all small cost NP structures. However, we know that except for characteristic function χ, IDP supports all the rest of GGF_k. So, we only need to show that χ can be written in terms of other arithmetical functions. This is easy to show: $\chi[\phi] \equiv \Sigma\{1 : \phi\}$. ∎

3.4.2 Non-expressibility Results under Complexity Assumptions

This part considers some natural arithmetical problems and shows that they cannot be encoded using only built-in arithmetic of ASP languages or the input language of the IDP system. Two such problems are considered: the Integer Factorization problem (which was defined in Example 3.4) and the Quadratic Residue Problem (which was defined in Example 3.5).

Various domain-restricted logical fragments

In this section, we first define several logical fragments by restricting their MX tasks. We also show that some practical KR system languages such as ASP and IDP fall into these fragments. Then, by showing some non-expressibility results for these logical fragments, we effectively prove that system languages of ASP and IDP cannot express integer factorization and quadratic residue problems naturally.

The first and most basic category of logics that we consider contains only those logics in which the MX task cannot increase the active domain:

Definition 3.8 (Active-domain-restricted logical fragment) *Let $\mathcal{L}$ be a logical fragment so that for all specifications ϕ in $\mathcal{L}$, and for all arithmetical structures $\mathcal{A}$ over σ (instance vocabulary) and all expansions of $\mathcal{A}$ such as $\mathcal{B}$ satisfying ϕ, we have that $adom_{\mathcal{B}} = adom_{\mathcal{A}}$. We call this logical fragment an* active-domain-restricted logical fragment.

The system language of IDP is an example of one of the logical specification languages that (under a natural condition) is active-domain restricted.

Proposition 3.1 *IDP system language without compact domain representation is an active-domain-restricted logical fragment.*

Proof: All predicates and functions in IDP system language should have proper type names. Also, all types are given as part of the input and the exclusion of compact domain representation ensures that types cannot be formed through compact representations of ranges that depend on values in the instance structure. Thus, the active domain of all structures satisfying a specification in IDP is exactly the active domain of the instance structure. ∎

Our second category is a more inclusive category that allows active domain to be expanded in the expanded model of a formula but limits this expansion in active domain to the value of a polynomial time computable function applied on the active domain of the instance structure.

Definition 3.9 (Polytime domain-restricted logical fragment) *Let $\mathcal{L}$ be a logical fragment so that for any specification ϕ in $\mathcal{L}$, there is a monotone polytime computable mapping $P : 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ with the following property: For all arithmetical structures $\mathcal{A}$ over σ (instance vocabulary) and for all structures $\mathcal{B}$ expanding $\mathcal{A}$ and satisfying ϕ, we have $adom_{\mathcal{B}} \subseteq P(adom_{\mathcal{A}})$. We call such logical fragments a* polytime domain-restricted logical fragment.

The two propositions that follow give some examples of polytime domain-restricted logical fragments. Firstly, we know that every active-domain-restricted logic is also polytime domain-restricted (thus, e.g., the language of IDP is polytime domain-restricted as well). Secondly, we give the example of ASP language of Gringo and Lparse (under some restrictions) as a crisp example of a polytime domain-restricted logic.

Proposition 3.2 *All active-domain-restricted logics are also polytime domain-restricted.*

Proof: Just take the polytime function P to be identity function. ∎

Proposition 3.3 *If we restrict the language of ASP accepted by Gringo and Lparse to programs that do not use summation aggregate Σ and compact domain representations, then the resulting language is a polytime domain-restricted logical fragment.*

Proof: Since, the programs are guaranteed not to use compact domain representations or summations, the size of ground versions of logic programs can only depend on the size of instance structure's active domain and not on the value of elements in the active domain.
For Gringo, we know that a correctly constructed ASP program should be level-restricted. So, we use induction on the levels of an ASP program and show that, for each level l, there is a monotone polytime program $P_l : 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ that, given an upper bound on the active domain of a structure that contains all predicates up to level $l-1$, P_l generates an upper bound on the active domain of a structure that contains all predicates up to level l. So, as any fixed ASP specification has only constantly many different levels, we can combine all P_l's to obtain a new monotone polytime program P that satisfies the polytime domain-restrictedness condition.
For Lparse, we just use the fact that all Lparse programs are also Gringo programs [79]. ∎

The condition on not including the summation operator Σ in the statement of Proposition 3.3 is essential because, if we allow summation, we will be able to describe predicates that consist of exponentially many values. The following ASP program shows one such scenario:

$$\begin{aligned} &V(2^i), \text{ for } i \in \{0, 1, 2, \cdots, n-1\}. \\ &A(X) \leftarrow \text{not } A'(X), V(X). \\ &A'(X) \leftarrow \text{not } A(X), V(X). \\ &E(X) \leftarrow X = \Sigma(Y; A(Y), V(Y)). \end{aligned} \tag{3.1}$$

This program has domain size n, but predicate E has domain 0 to $2^n - 1$.

The last and most inclusive category of logics that we discuss here is called loosely domain-restricted logics. In this category as well, active domain cannot be expanded arbitrarily, however, this category allows exponential size increase in some cases.

Definition 3.10 (Loosely domain-restricted logical fragment) *Let $\mathcal{L}$ be a logical fragment so that for any specification ϕ in $\mathcal{L}$, there is a function $f : \mathbb{N} \to \mathbb{N}$ and a monotone polytime mapping $P : 2^{\mathbb{N}} \times 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ such that: For all arithmetical structures $\mathcal{A}$ over σ (instance vocabulary) and for all structures $\mathcal{B}$ expanding $\mathcal{A}$ and satisfying ϕ, we have $adom_{\mathcal{B}} \subseteq P(adom_{\mathcal{A}}, \{1, 2, 3, \cdots, f(|adom_{\mathcal{A}}|)\})$. We call such logical fragments a* loosely domain-restricted logical fragment.

As before, every polytime domain-restricted logical fragment is also a loosely domain-restricted logical fragment with f being a constant function and P being the same as before except it takes two arguments and neglects the second one.

Proposition 3.4 *All polytime domain-restricted logics are also loosely domain-restricted.*

Proof: Let P be the function that witnesses polytime domain-restrictedness of a logic $\mathcal{L}$. Define $f'(n) := 1$ and $P'(S, T) := P(S)$. Together, f' and P' witness that $\mathcal{L}$ is also loosely domain-restricted. ∎

Note that, while all polytime domain-restricted logics are also loosely domain-restricted, there is an important distinction between them. Namely, when we fix the specification, polytime domain-restricted logical fragments are guaranteed to have polytime grounding (in the size of instance structure). However, for loosely domain-restricted logical fragments, such grounding cannot be guaranteed.

As a crisp example of some languages that are loosely domain-restricted but not polytime domain-restricted, we present the language of ASP with summation (but without compact domain representation).

Proposition 3.5 *If we restrict the ASP language of Gringo and Lparse so that programs cannot use compact domain representations, the resulting language is a loosely domain-restricted logical fragment.*

Proof: Again, since compact domain representation is not allowed, the size of active domain of expanded models can only depend on the size of active domain of the instance structure and not on the values in the instance structure. Note that, unlike in the case of Proposition 3.3, here, the size of new active domain can be exponentially big. Equation (3.1) shows one such case that summation

introduces exponentially many possible values. The important point here is that the size of this set can be bound by a function that only depends on the size of input structure and is independent of values in the input structure, i.e., a function $f : \mathbb{N} \mapsto \mathbb{N}$ exists such that if the size of instance structure's active domain is n, the size of expansion structure's active domain would always be less than $f(n)$ no matter how big or small the numbers in instance structure are.

For Gringo, again, we use the level-restrictedness property. Assume that your specification ϕ has l levels. As discussed above, each summation can (potentially) exponentiate the size of the domain. But, by level-restrictedness property, summations seperate levels. Thus, at most l different exponentiations can occur. Now, if we take f as follows, it gives an upper bound on the number of elements at level l:

$$f(n) = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}}_{l\ times}$$

Then, we use the same monotone polytime program P as in Proposition 3.3 with the difference that it takes two arguments and neglects the second one. This program runs in polytime because the size of its input is so large that all of its computation time is bounded by a polynomial in that (large) number. Please note that the function f given above is a very rough upper bound and we believe that upper bounds of form $f(n) = 2^{poly(n)}$ work too (although with a more detailed proof).

For Lparse, again, we only use the fact that all Lparse programs are also Gringo programs [79]. ∎

Non-expressibility for domain-restricted logical fragments

We now prove that integer factorization and quadratic residue problem cannot be naturally axiomatized in the logical fragments we defined. For active-domain-restricted logical fragments, this is obvious. We need new numbers except those in the domain. Below, we prove the same property for the two other domain-restricted fragments, i.e., polytime and loosely domain-restricted logical fragments.

Theorem 3.2 *If $\mathcal{L}$ is a polytime or loosely domain-restricted logical fragment, then*

1. *$\mathcal{L}$ cannot express integer factorization using built-in arithmetic unless factorization is in polytime.*
2. *$\mathcal{L}$ cannot express quadratic residue problem using its built-in arithmetic unless P=NP.*

Proof:(for polytime domain-restricted logical fragments) Let $\mathcal{L}$ be a polytime domain-restricted logical fragment and ϕ be a specification in such a fragment. Then, by definition, there is a monotone

polytime program P that gives an upper bound on the active domain of all valid expansions of a given instance structure.

Now, if ϕ axiomatizes integer factorization with its built-in arithmetic, at least one factor of number n should appear in the output of P. So, checking all numbers that P outputs gives us one such factor. Also, as P is polytime, the whole checking procedure would also be polytime and we would solve integer factorization in polytime.

Similarly, if ϕ axiomatizes quadratic residue problem using its built-in arithmetic, then x appears in the output produced by P. So, checking all outputs of P gives us such x if it exists. Again, P being polytime implies that the whole procedure is also polytime and P=NP (because quadratic residue problem is NP-complete). ∎

Proof:(for loosely domain-restricted logical fragments) Let $\mathcal{L}$ be a loosely domain-restricted logical fragment and ϕ be a specification in such a fragment. Then, by definition, there is a function f and monotone polytime program P such that the active domain of all valid expansions $\mathcal{B}$ of an input structure $\mathcal{A}$ are upper-bounded by $P(adom_{\mathcal{A}}, \{1, 2, 3, \cdots, f(|adom_{\mathcal{A}}|)\})$.

However, in the case of the two problems above (factorization and quadratic residues), we know that the number of elements in the input structure is always constant (one element in factorization and 3 elements in quadratic residue). Therefore, $f(|adom_{\mathcal{A}}|)$ is always a constant (depending on f but not $\mathcal{A}$). Thus, the set $\{1, 2, 3, \cdots, f(|adom_{\mathcal{A}}|)\}$ does not depend on structure $\mathcal{A}$. So, program P takes the active domain of $\mathcal{A}$ and returns a set S which upper-bounds the active domain of all valid expansions of $\mathcal{A}$. The rest of the proof can be carried out in the same way as the previous case. ∎

Corollary 3.3 *Using only built-in arithmetic of ASP language of Gringo and Lparse (with or without Σ) and without using compact domain representations,*
(1) Unless P=NP, the quadratic residue problem cannot be axiomatized naturally, and
(2) Unless integer factorization is in P, it cannot be naturally axiomatized.

So, we proved that the two problems of factorization and quadratic residue cannot be axiomatized in either ASP or IDP system languages without compact domain representations. Indeed, compact domain representation can be used to naturally axiomatize both problems above. However, there are two drawbacks associated with using compact domain representations to axiomatize such problems:

1. Once we include compact domain representation as an operator, it is impossible to limit the expressiveness of a language to any reasonably feasible complexity class (such as P, NP, Σ_k^P for some k or even PSPACE). In fact, in the presence of compact domain representation,

one can easily describe, for example, the NEXP-complete problem of tiling. Therefore, not only would any hope for polynomial time grounding be lost, but one cannot even hope for semi-efficient solving.

2. Moreover, compact domain representation is not really an option for any reasonable size domain. For example, in the case of factorization, the currently interesting numbers to factor in practice use more than 200 digits. So, using compact domain representation, one is left with a domain of size about 10^{200} that cannot be stored in any physically available memory (because the domain size is more than the number of atoms in the observable universe).

3.4.3 Uncoditional Inexpressibility Results

Previously, our inexpressibility results were conditional on not using compact domain representation and also on complexity results. In this section, we lift these conditions and show that there exist arithmetical problems in NP (and even in P) that Gringo and Lparse cannot express over arithmetical structures without product aggregate even if P=NP and compact domain representation is allowed. So, since product aggregate is in fact not included in the language of Gringo and Lparse, it follows that they cannot express (under any conditions) such problems.

What we prove in this section is that all numerical terms in an ASP program expressed in the system language of Gringo or Lparse can be represented by a linear number of bits in the binary representation size of the maximum number in the active domain of the instance structure.

Theorem 3.3 (Number Limit Theorem) *The binary representation size of any evaluations of a term in ASP or Gringo has at most linear size in the size of binary representation of the maximum number in the active domain of instance structure, i.e. for each ASP program P there exists a function $f : \mathbb{N} \mapsto \mathbb{N}$ such that for all terms $t(\bar{x})$ used in P, all instance structures $\mathcal{A}$, all satisfying expansions $\mathcal{B}$ of $\mathcal{A}$, and all assignments from $\bar{x}$ to active domain of $\mathcal{B}$, we have that: $||t(\bar{a})|| \leq f(|adom(\mathcal{A})|).|| \max\{adom(\mathcal{A})\}||$.*

Proof: We use an induction on first the levels of an ASP program and then the structure of well-formed terms. In this proof, we use M for the maximum number in the active domain of the instance structure and m for the binary encoding size of M. We also use n to denote the size of the domain.

- By level-restrictedness property, all variables x are guarded by one of the following: (1) an assignment $x = t'(\bar{y})$ where $t'(\bar{y})$ does not depend on x directly or indirectly, (2) a positive literal in the body with a level less than the level of the predicate symbol in the head, or (3) a

compact domain guard. Thus, the value of a variable is bound by that of another term or the values in the active domain.

- All constants c satisfy this property because $m \geq 1 \Rightarrow c \leq c \times m$.
- For a term $t(\bar{x})$ of form $t_1(\bar{x}) + t_2(\bar{x})$, we have:

$$||t(\bar{x}|| \leq ||t_1(\bar{x}|| + ||t_2(\bar{x}|| \leq f_1(n).m + f_2(n).m \leq (f_1(n) + f_2(n)).m$$

- For a term $t(\bar{x})$ of form $t_1(\bar{x}) \times t_2(\bar{x})$, we have:

$$||t(\bar{x})|| \leq ||t_1(\bar{x})|| + ||t_2(\bar{x})|| + 2 \leq (f_1(n) + f_2(n) + 2).m$$

- For the aggregates $\min$ and $\max$, we know that the resulting term is bounded by the same inequality that bounds the inner term.
- The count aggregate is a special case of a sum aggregate. So, we only consider the case of a summation aggregate $t(\bar{x}) := \Sigma_{\bar{y}}(t'(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y}))$. For such a $t(\bar{x})$, we have: $t(\bar{x}) \leq k \times M'$ where k upper bounds the number of tuples satisfying $\phi(\bar{x}, \bar{y})$ and M' upper bounds the value of $t'(\bar{x}, \bar{y})$ under condition $\phi(\bar{x}, \bar{y})$.

  Moreover, by induction hypothesis, we know that $||t'(\bar{x}, \bar{y})|| \leq f'(n) \times m$. So, $M' := M^{f'(n)}$ is a good upper bound on $t'(\bar{x}, \bar{y})$ because: $t'(\bar{x}, \bar{y}) \leq 2^{m.f'(n)} = M^{f'(n)} = M'$. Also, as all variables $\bar{x}$ and $\bar{y}$ are guarded, we know that for each variable $z \in \bar{x} \cup \bar{y}$, it is bound by either the values in the domain or the values of some other term, i.e., we have $f_z(n)$ such that all possible values a of z satisfy $||a|| \leq f_z(n) \times m$. Therefore, $a \leq 2^{m.f_z(n)} = M^{f_z(n)}$ for all possible values a of z. Thus, z can only get $M^{f_z(n)}$ different values.

  Now, let $f''(n) := \Sigma_{z \in \bar{x} \cup \bar{y}} f_z(n)$. Now, we can set k to be $M^{f''(n)}$. This is clearly an upper bound on the number of different possible assignments to $\bar{x} \cup \bar{y}$ because:

$$\begin{aligned} |\{(\bar{x}, \bar{y}) \mid \phi(\bar{x}, \bar{y})\}| &\leq \Pi_{z \in \bar{x} \cup \bar{y}} |\{a \ : \ \text{a sat. assignment } \tau \text{ of } \phi \text{ assigns } a \text{ to } z\}| \\ &\leq \Pi_{z \in \bar{x} \cup \bar{y}} M^{f_z(n)} \\ &= M^{\Sigma_{z \in \bar{x} \cup \bar{y}} f_z(n)} \\ &= M^{f''(n)} = k \end{aligned}$$

  Thus, we have $t(\bar{x}) \leq k \times M' = M^{f'(n)} \times M^{f''(n)} = M^{f'(n)+f''(n)}$. So, $||t(\bar{x})|| \leq (f'(n) + f''(n)) \times m$.

■

Theorem 3.3 suggests that, even in the presence of compact domain representations and even if P=NP, there are still arithmetical problems in NP that cannot be represented in the ASP language of Gringo and Lparse. One such problem is the prime factor permutation problem from Section 3.1. Recall that the expansion structure in this problem sometimes needs to represent integers whose binary encoding size is quaratic in the binary encoding size of the only number in the active domain (which is also the maximum number in the active domain). The following corollary gives us to concrete examples:

Corollary 3.4 *ASP language of Gringo and Lparse cannot describe the problems below:*
(1) The problem of finding k such that $k = n^{\lceil \log n \rceil}$ (for number n given in the input).
(2) The prime factor permutation problem of Example 3.6.

Proof: Both of these problems have the property that the number of bits needed to represent the output value is not bound by any function that is linear in the number of bits needed to represent the maximum number in the input. ■

Note that both inexpressible problem of Corollary 3.4 are arithmetical NP search problems. In fact, the first problem is solvable in P. These examples show a deep incompatibility between the complexity of solving problems that are described in ASP and the expressiveness of the language. That is, although we know that some NEXP-complete problems are axiomatizable in the language of ASP and in the presence of compact domain representation, this knowledge does not tell us anything about how expressive our language is, i.e., there might exist problems such as those in Corollary 3.4 (especially the first one that is even polytime computable) that are of far far less complexity and yet not axiomatizable. Thus, we here want to further express our concern about *properly studying how expressive our specification languages are*.

3.4.4 Safety in ASP

In Subsection 3.4.1, we defined the notion of λ-restricted ASP programs and focused on only this fragment of ASP. However, the syntax of ASP solvers such as gringo has expanded and now allows specifications beyond λ-restricted programs. In [80], the authors have defined this more extended syntax which is known as safe programs. Under the safe program syntax for ASP being λ-restricted is no longer needed. In fact, this fragment of ASP programs allows specifications with infinite groundings for which the grounder cannot be guaranteed to stop. For example, the authors of [80] axiomatize the behaviour of a universal Turing machine as a safe Gringo specification.

Similar to the authors of [80], we also believe that, for a specification language, being expressive is a virtue in its own right and that having a Turing-complete solver language can prove itself useful in cases where a computationally very complex problem needs to be axiomatized. But, in here, we want to briefly discuss two important issues that have not yet been addressed for this extended language of gringo:

1. **Complexity of Fragments of the Language:** As we discussed before, through studying the expressiveness of a language and its fragments, one can identify syntactic conditions which guarantee that if the description of a problem falls within those conditions, then some known bounds on the resources needed for solving that problem can be inferred automatically. This way, even the naive users without any knowledge of complexity theory can try to specify their intended problem using the most limited syntactical fragment possible and thus avoid high-complexity solving mechanisms. While such studies are abundant for the core language of ASP, i.e., rules of different forms, they are still missing with respect to the language of practical solvers such as Gringo and also with respect to the inclusion of arithmetical statements in ASP in general.

2. **Completeness vs. Capturing:** As discussed in Section 3.1, studying the completeness (or hardness) of a language with respect to a complexity class is not equivalent to studying the capturing property. For example, for the extended language of Gringo according to [80], we know that this language is Turing-complete. However, such a knowledge does not imply the expressibility of all Turing-computable problems within this language. For the sake of comparison, consider the example of λ-restricted ASP programs which could define NP-complete arithmetical problems but which could not specify the integer factorization problem (i.e., completeness did not imply capturing). In the same manner, being Turing-complete does not imply that every Turing-computable arithmetical problem is expressible as safe Gringo program. We still need to study which problems are expressible in Gringo.

We hope that this chapter will be useful in addressing these issues for the language of ASP and for the complexity class NP. Although the capturing results of this chapter are for first-order specifications, we believe that, with minor efforts, these results can be extended to ASP as well. Therefore, we hope that one can identify a fragment of ASP language that captures the class of arithmetical NP search problems, i.e., arithmetical problems described within this fragment only need the computational power of NP and that this fragment is universal for arithmetical NP search problems.

3.5 Logic PBINT

In this section, we describe a logic that can unconditionally characterizes NP problems involving arithmetic. The first step towards this goal is to use a different background structure:

Definition 3.11 *A* Compact Arithmetical structure *is a structure $\mathcal{N}^c$ having at least $(\mathbb{N}; 0, 1, +, \times, <, ||\ ||)$ with domain $\mathbb{N}$, the natural numbers, where 0, 1, $+$, $\times$ and $<$ have their usual meaning and $||x||$ returns the size of binary encoding of number x, i.e., $||x|| = 1 + \lfloor log_2(x+1) \rfloor$. Other functions, predicates, and multi-set operations (min, max etc.) may be included, provided every function and relation of $\mathcal{N}^c$ is polytime computable.*

Requirements on σ

As before, we consider embedded MX, but the embedding is into the compact arithmetical structure. We make some assumptions about the instance vocabulary σ. It contains predicate $adom_{\mathcal{A}}$ and a constant $SIZE$. The constant $SIZE$ is equal to $|adom_{\mathcal{A}}| \times S$ where $|adom_{\mathcal{A}}|$ is the number of elements in the active domain and S is the size of binary encoding of the maximum element of the active domain. In other words, $SIZE$ upper-bounds the number of bits needed to encode (in binary) the input structure $\mathcal{A}$ embedded in $\mathcal{N}^c$. We also need a constant *default* denoting a particular default value needed in upper guards on functions. Its meaning is specified by the user.

Logic PBINT

We introduce a new logic, PBINT, standing for Polynomially Bounded Integers. This logic is a variant of the double-guarded logic except we use compact arithmetical structures and allow function symbols in both σ and ε, or new kinds of guards, with more freedom in existential and upper guards on the outputs of expansion functions. The three forms of guards in PBINT are as follows:

1. **Instance Guards** are instance relations (including $adom_{\mathcal{A}}$) interpreted by the instance structure $\mathcal{A}$. Note that, although not required to be so, all specifications can be rewritten so as they only use $adom_{\mathcal{A}}$ as an instance guard.
2. **Polynomial Range Guards** are relations of the form $poly_1(SIZE) \leq x \leq poly_2(SIZE)$ with $poly_1$ and $poly_2$ two polynomials.
3. **PBINT Guards** are relations of form $||x|| \leq poly(SIZE)$ where $poly(SIZE)$ is a polynomial depending only on the constant $SIZE$.

Instance guards and polynomial range guards define ranges of size at most polynomial in the binary encoding size of structure. However, PBINT guards can define ranges with exponentially many different integers. For example, condition $||x|| \leq SIZE$ is equivalent to $x < 2^{SIZE-1}$ which is exponential in the value of $SIZE$. Also, note that guards definable by stratifiable inductive definitions with (1), (2) as the base cases can be added without changing our results.

Definition 3.12 (logic PBINT) *We define our logic as follows.*

Background Structure: *the compact arithmetical structure.*

Terms *are constructed as usual over $\nu \cup \sigma \cup \varepsilon$ with ν being the background vocabulary, σ the instance vocabulary, and, ε the expansion vocabulary.*

Formulas:

(a) **Upper Guards**

i. Expansion relations *are upper-guarded by instance or polynomial range guards.*

ii. An expansion function *f has an upper guard of form $\forall \bar{x} \forall y\ (f(\bar{x}) = y \Rightarrow (G(\bar{x}, y) \vee y = default))$ where $G(\bar{x}, y)$ is a conjunction of guards jointly guarding variables $\bar{x}$ and y so that $\bar{x}$ is upper-guarded by instance or polynomial range guards and y is upper-guarded by any of the three types of guards.*

(b) **Lower Guards**

i. Existential guards: any of the three types of guards.

ii. Universal guards: instance or polynomial. range guards.

The constant "default" can be interpreted by any number at the user's choice. The part $y = default$ in *(a(ii))* above is needed because all functions in FO logic are total, thus defined on all natural numbers. Without that part, the upper guard axioms on expansion functions would always be false making all specifications with such functions useless.

Functions have meaningful (non-default) outputs on a finite number of inputs. Thus, we can obtain a finite representation (encoding) of instance and expansion structures.

Having functions in the instance vocabulary imposes a small inconvenience with the definition of the active domain. In a formalism based on classical logic, all terms are total, and therefore defined on all integers. Thus, if we add functions, the notion of an active domain becomes meaningless. On the other hand, the user might (quite reasonably) assume that the inputs and the outputs of the instance functions are from a finite domain, which makes these functions to be non-total. A safe solution would be to advise the user to use graphs of functions (i.e., the corresponding predicates) instead of instance functions. It would solve the problem with the definition of an active domain,

but this solution seems too limiting for a nice logic. Instead, we choose to allow instance functions, but to require all instance functions to have upper guards and the "default" value for inputs outside of the intended range, just as we have for expansion functions. Active domain now contains all elements of the universe contained in all instance relations, together with all elements in the ranges of the instance functions.

Capturing NP with PBINT-MX

Theorem 3.4 *Let $\mathcal{K}$ be an isomorphism-closed class of compact arithmetical embedded structures over vocabulary σ. Then the following are equivalent:*

1. $\mathcal{K} \in NP$,
2. *there is a PBINT sentence ϕ of a vocabulary $\tau = \sigma \cup \nu \cup \varepsilon$, such that $\mathcal{A} \in \mathcal{K}$ iff there exists an expansion $\mathcal{B}$ of $\mathcal{A}$ with $\mathcal{B} \models \phi$.*

The importance of this theorem is in its ability to capture all NP with arithmetic. As shown previously, most of previous frameworks for arithmetic in KR suffer from the fact that they can only axiomatize certain arithmetical problems in NP. For example, we showed that ASP languages and the language of the IDP system cannot axiomatize integer factorization using their built-in arithmetic. On the other hand, Theorem 3.4 shows that PBINT can axiomatize exactly those problems involving arithmetic which are in NP.

Moreover, the background structure of PBINT is much simpler than many background structures in practical KR languages. In particular, there is no built-in aggregate in compact arithmetical structures. Together with Theorem 3.4, it shows that PBINT can define aggregates in terms of more primary operations (those given in compact arithmetical structures). Therefore, adding aggregates to your language would not increase the expressibility of your language.

Thus, PBINT gives you a very concrete basis to build your language upon. It tells you that if your language somehow supports all PBINT constructs, you can (1) be sure that your language captures all of NP and (2) unless your other constructs are very powerful (outside NP $\cap$ co-NP), you can safely add them to your language without worrying about its complexity implications.

The proof for the two different directions of this theorem are given in separate subsections. But, first, we introduce a characterization for PTIME due to Bellantoni and Cook [22] which is needed for our proof of (1) $\Rightarrow$ (2).

3.5.1 Bellantoni-Cook Characterization of PTIME

We briefly describe a functional language introduced by Bellantoni and Cook [22] which captures polytime functions. It has originally been defined to work on strings in $\{0,1\}^*$. But, as such strings encode numbers, we have reformulated the operations in numerical terms.

Functions in Bellantoni-Cook form have two sets of parameters separated by a semicolon. Parameters to the left of semicolon are called "normal" inputs and those to its right are "safe" inputs. This separation disables the possibility of introducing recursions whose depth depend on the result of other recursions. This property is essential to prove that such functions are poly-time computable. Here are the constructs:

1. Zero: $Z(;) = 0$.
2. Projections $\pi_j^{n,m}(x_1, \cdots, x_n; x_{n+1}, \cdots, x_{n+m}) = x_j$.
3. Successors $S_0(;a) = 2 \times a,\ S_1(;a) = 2 \times a + 1$.
4. Modulo 2: $M(;a) = a \bmod 2$.
5. Predecessor: $P(;a) = \lfloor \frac{a}{2} \rfloor$.
6. Conditional: $C(;a,b,c) = \textbf{if } 2|a \textbf{ then } b \textbf{ else } c$.
7. Safe recursion that defines $n+1$-ary function f based on n-ary function g and the $n+2$-ary functions h_0 and h_1:

$$\begin{aligned} &f(0, \overline{x}; \overline{y}) = g(\overline{x}; \overline{y}), \\ &f(2a, \overline{x}; \overline{y}) = h_0(a, \overline{x}; \overline{y}, f(a, \overline{x}; \overline{y})), \\ &f(2a+1, \overline{x}; \overline{y}) = h_1(a, \overline{x}; \overline{y}, f(a, \overline{x}; \overline{y})). \end{aligned}$$

8. Safe composition that defines function f based on functions $r_0, \cdots, r_{k+k'}$:

$$\begin{aligned} &f(\bar{x}; \bar{y}) = \\ &\quad r_0(r_1(\bar{x};), ..., r_k(\bar{x};); r_{k+1}(\bar{x}; \bar{y}), ..., r_{k+k'}(\bar{x}; \bar{y})). \end{aligned}$$

This language interests us as it is a purely syntactic characterization of PTIME which is based on numbers. Furthermore, the language is free of any unnatural functions for bounding growth of numbers.

Bellantoni-Cook's theorem says that any function defined in this form is PTIME and that for any PTIME computable function $f(\overline{a})$, there is a function $f'(w; \overline{a})$ such that $f(\overline{a}) = f'(w; \overline{a})$ for all a's and for all w's satisfying $||w|| \geq p_f(\overline{||a||})$ (where $||x||$ is the binary encoding size of x and p_f is a polynomial depending on f and constructible based on Bellantoni-Cook's proof).

3.5.2 NP $\subseteq$ PBINT MX

Proof: Let us first review our proof structure for (1) $\Rightarrow$ (2).

1. We know NP problems have PTIME verifiers.
2. By Bellantoni and Cook's theorem, every such polytime verifier can be given in their syntax.
3. So, it remains to show that, verifier V in Bellantoni-Cook form, can be turned into axiomatization ϕ in PBINT so that for σ-structure $\mathcal{A}$, ϕ is satisfiable by an expansion of $\mathcal{A}$ iff there is a polysize certificate for $\mathcal{A}$ accepted by V.

As this proof is so detailed, we only include the proof idea here. The full proof is in the Appendix. The proof constructs PBINT specification ϕ based on verifier V. In ϕ, expansion vocabulary ε consists of:

1. $B : \mathbb{N} \times \mathbb{N}$ to map start times to functions, e.g., $B(5, c_f)$ means that, at time 5, function f has started running (c_f is a constant used to refer to function f).
2. $E : \mathbb{N} \times \mathbb{N}$ which, similarly, maps start times to end times, e.g., $E(5, 10)$ means that the function that had started running at time 5 ended running at time 10, and together with $B(5, c_f)$, it means that function f started at time 5 and ended at time 10.
3. $r : \mathbb{N} \to \mathbb{N}$ is used to store result of function executions. It is needed only when execution of a function terminates. For instance, continuing example above, having $r(10) = 2$ means that result of computing function f is 2 (because it was f that ended at time 10).
4. $arg : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is used for storing function arguments. Semantically, $arg(n, m) = k$ means that the m^{th} argument of function starting at time n is k. For example, having $arg(5, 1) = 7$ together with examples listed in items (1) to (3), means that function f started running and at time 5 on argument 7, and it finished at time 10 and gave result 2, so $f(7) = 2$.

This expansion vocabulary enables us to simulate the behavior of a program in Bellantoni-Cook form. We will have axioms saying what each function does. For example, for base function Z we have that it finishes immediately and gives zero as result. This is axiomatized as below:

$$\forall n\ (T(n) \wedge B(n, c_Z) \Rightarrow E(n, n + 1)),$$
$$\forall n\ (T(n) \wedge B(n, c_Z) \Rightarrow r(n + 1) = 0).$$

where $T(n)$ is a lower guard which bounds the time needed for simulating verifier V (a polynomial in the size of binary encoding of structure).

All other base functions have similarly simple axiomatizations. Functions defined in terms of other functions (using safe recursion or safe composition) need more complex (but still straightforward) axiomatizations. All these details can be found in the full proof.

There are some more subtle issues that have to be addressed in order to give a correct proof, e.g. encoding structures as numbers. All these subtleties are dealt with in the full proof given in the Appendix (which did not fit here, please see the online version). ∎

3.5.3 PBINT MX⊆ NP

Proof: Here, the proof goes as follows:

1. We first show that, given an MX specification ϕ in PBINT, you can find equivalent ψ in ∃SO by using binary encodings of numbers.
2. Next, we show that structure $\mathcal{A}$ for ϕ is convertible to structure $\mathcal{A}'$ for ψ (in polytime) so that satisfying expansion $\mathcal{B}$ of $\mathcal{A}$ exists iff satisfying expansion $\mathcal{B}'$ of $\mathcal{A}'$ exists.
3. Then, by Fagin's theorem, PBINT MX⊆NP.

To obtain ψ, we first create a specification in which all existentially quantified variables with PBINT guard G are replaced by skolemized PBINT functions upper-guarded by G. Then, this specification is converted to ψ by replacing PBINT functions with relations that encode value of the function in binary. For example, for PBINT function $f(x_1, \cdots, x_n)$, relation $Q_f(x_1, x_2, \cdots, x_n, k)$ is introduced with k being guarded by new relation R. The idea is that $Q_f(x_1, \cdots, x_n, k)$ holds iff the k-th bit of binary encoding of $f(x_1, \cdots, x_n)$ is one.

We know that all numbers in a PBINT specification are guarded. Hence, there is a polynomial $p(n)$ such that $2^{p(SIZE)}$ is greater than all numbers generated in ϕ. So, assuming that we have a relation R containing all numbers $0, \cdots, p(SIZE)$, describing operations of background structure is easy. For example, assuming that x, y and z are encoded by unary predicates Q_x, Q_y and Q_z, the relation $x = y + z$ can be axiomatized as follows:

$$
\begin{aligned}
&\forall k\ (R(k) \Rightarrow \\
&\quad (Carry(k) \Leftrightarrow (Q_x(k) \Leftrightarrow (Q_y(k) \Leftrightarrow \neg Q_z(k))))), \\
&Carry(k) := \\
&\quad \exists k'\ (R(k') \wedge k' < k \wedge Q_y(k') \wedge Q_z(k') \wedge CF(k, k')), \\
&CF(k, k') := \\
&\quad \forall k''\ (R(k'') \wedge k' < k'' < k \Rightarrow Q_y(k'') \vee Q_z(k'')).
\end{aligned}
$$

Although cumbersome, all other background operations can be similarly axiomatized.

Now, structure $\mathcal{A}'$ is obtained from $\mathcal{A}$ by adding unary relation R to $\mathcal{A}$ and converting all numbers in $\mathcal{A}$ to their binary representation. These tasks can be done in polytime and so obtaining $\mathcal{A}'$ from $\mathcal{A}$ is polytime achievable.

So, as ψ is in $\exists$SO, the task of model expansion for ψ is in NP. Also, as $\mathcal{A}$ is polytime convertible to $\mathcal{A}'$ and existence of a satisfying expansion for $\mathcal{A}$ is equivalent to existence of a satisfying expansion for $\mathcal{A}'$, model expansion for ϕ will also be in NP. ∎

3.6 PBINT as the Basis for a Modeling Language

In Section 3.5, we introduced the logic PBINT. In this section, we informally introduce a modelling language (called PBINT-Spec) which is based on the logic PBINT and has the beautiful syntactical constructs of a practical modelling language. For example, PBINT-Spec supports finite types as well as integers. Moreover, as we will see through the examples, explicit upper guard and lower guard axioms of PBINT logic are no longer needed in PBINT-Spec as they will be replaced by types. This way, a specification in the PBINT-Spec becomes more readable than its counterpart in logic PBINT (without affecting the expressiveness of the language). The main objective of this section is to demonstrate that arithmetical NP search problems can be naturally represented in a language based on PBINT. The two examples of integer factorization and quadratic residue are also included in this section so as the reader could contrast them against the results of Section 3.4.

The first example in this section gives both the axiomatization in PBINT logic and the axiomatization in the PBINT-Spec. This way, the reader can compare the two axiomatizations. The rest of the examples in this section will only give the specification in PBINT-Spec. Therefore, the first example also serves as a reference point for translating other specifications in our modeling language to axiomatizations in PBINT logic (if the need arises).

Example 3.8 (Disjoint Scheduling) *Given a set of Tasks, $t_1, \cdots, t_n$ and a set of constraints, find a scheduling 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 is also two predicates $P(t_i, t_j)$, that says task t_i should end before task t_j starts, and $D(t_i, t_j)$, which means that the two tasks t_i and t_j cannot overlap. We are asked to find two functions $start(t_i)$ and $end(t_i)$ satisfying the given conditions.*

In PBINT logic, we axiomatize this problem as follows: Instance vocabulary σ consists of symbols EST, LET, L, $Task$, P and D. Expansion vocabulary consists of two functions $start$ and end. The axiomatization below first gives the upper guards on these functions and then the axioms:

$$
\begin{aligned}
&\forall t \forall s\ (start(t) = s \Rightarrow (Task(t) \wedge ||s|| \leq SIZE) \vee s = default),\\
&\forall t \forall e\ (end(t) = e \Rightarrow (Task(t) \wedge ||e|| \leq SIZE) \vee e = default),\\
&\forall t_i\ (Task(t_i) \Rightarrow start(t_i) \geq EST(t_i)),\\
&\forall t_i\ (Task(t_i) \Rightarrow end(t_i) \leq LET(t_i)),\\
&\forall t_i\ (Task(t_i) \Rightarrow start(t_i) + L(t_i) = end(t_i)),\\
&\forall t_i \forall t_j\ (P(t_i, t_j) \Rightarrow end(t_i) \leq start(t_j)),\\
&\forall t_i \forall t_j\ (D(t_i, t_j) \Rightarrow end(t_i) \leq start(t_j) \vee end(t_j) \leq start(t_i)).
\end{aligned}
\tag{3.2}
$$

In PBINT-Spec, upper and lower guards are defined by types and need not be given explicitly. Here, the predicate $Task$ is a type and functions $start$ and end are functions from type $Task$ to integer type. So, predicate $Task$ can disappear from the above sentences. The result is as follows:

```
Given
  Types {
    Task : Finite enumerable,
    Time : PBINT(Size) }
  Relations {
    P : Task × Task,
    D : Task × Task }
  Functions {
    EST : Task ↦ Integer,
    LET : Task ↦ Integer,
    L : Task ↦ Integer }
Find
  Relations {
    Start : Task ↦ Time
    End : Task ↦ Time }
Such that
  ∀t:Task (Start(t) ≥ EST(t)).
  ∀t:Task (End(t) ≤ LET(t)).
  ∀t:Task (Start(t) + L(t) = End(t)).
  ∀t1,t2:Task (P(t1,t2) → End(t1) ≤ Start(t2)).
  ∀t1,t2:Task (D(t1,t2) → (End(t1) ≤ Start(t2) ∨ End(t2) ≤ Start(t1))).
```

One can easily see that the upper guard axioms in the PBINT Specification 3.2 are replaced by type "Time" which is a PBINT type and works as the output type of functions "Start" and "End". Therefore, here, one does not need to give upper guards explicitly. They can be implicitly derived by

the language. The same goes to lower guards: they are also replaced by types.

Example 3.9 (Integer Factorization) *Continuing Example 3.4, the integer factorization problem can be specified in PBINT-Spec as follows:*

```
Given
  Types {
    Factor : PBINT(Size) }
  Constants {
    n : Integer,
    c : Integer }
Find
  Constants {
    k : Factor }
Such that
  p > 1 ∧ p < n ∧ p ≤ c.
  ∃k':Factor (k × k' = n).
```

Example 3.10 (Quadratic Residues) *Continuing Example 3.5, the quadratic residue problem can be specified in PBINT-Spec as follows:*

```
Given
  Types {
    Num : PBINT(Size) }
  Constants {
    n : Integer,
    c : Integer,
    r : Integer }
Find
  Constants {
    s : Num }
Such that
  s ≥ 0 ∧ s ≤ c ∧ s < n.
  ∃k:Num (s × s = k × n + r).
```

3.7 Related Work

Research in databases over infinite structures can be traced back to the seminal paper by Chandra and Harel [35]. There are several follow-up papers with developments in several directions including [193, 179, 100], and more recent [99]. Topor [193] studies the relative expressive power of several query languages in the presence of arithmetical operations. He also investigates domain independence and genericity in such frameworks.

Another line of database-motivated work over infinite background structures is embedded model theory (See [132, 133]). Work in this area generally reduces questions on embedded finite models to questions on normal finite models. An important result in this area is the natural-domain-active-domain collapse for $\exists$SO for embedded finite models, as well as other deep expressiveness results. The work also describes a notion of safety (through e.g. range-restriction) to achieve safety with many background structures, and connections between safety and decidability. The active domain quantifiers are similar to our proposal of lower guards, however our goal was to reflect what is used in practical languages, namely the so-called domain predicates of Answer Set Programming and type information from other languages. We have done it through the use of upper and lower guards. In general, research in database theory is mostly focused around computability and the expressive power of query languages, while our interest, following [99] is in capturing complexity classes, but in connection with specification/modelling languages. We plan, however, to investigate the applicability of domain-independence, range-restrictedness and other notions from embedded model theory to practical modelling languages.

Grädel and Gurevich [100] studied logics over infinite background structures in a more general computer science context. They characterized NP for arithmetical structures under some small weight property, generalized to the small cost condition in [189] (see [189] for a more detailed discussion). While this condition corresponds to existing languages (as shown in Section 3.4.1), our work here gives an unconditional result for capturing NP in the presence of arithmetical structures, and thus is a step forward in the development of such languages. Instead of controlling access to the background structure through the use of weight terms [100] , we rely on guarded fragments, which is much closer to practical specification languages.

The work we mentioned so far is the closest to our proposal, and was the most inspirational. The research on descriptive complexity in the embedded setting also includes the work of Grädel and Meer [104], as well as Grädel and Kreutzer [103]. These works prove interesting results for real number arithmetic. So, they were not applicable to our current work on integer arithmetic. However,

they will be of immense importance for our future work on real number arithmetic. Another line (Cook, Kolokolova and others [43]) establishes connections between bounded arithmetic and finite model theory, in particular by relying on Grädel's characterization of PTIME. Since we needed a functional characterization of PTIME, this work was not suitable to use for our purposes.

Another direction on capturing complexity classes is bounded arithmetic, including [27, 176, 22]. However, the characterization of complexity classes there is in terms of *provability* in systems with a limited collection of non-logical symbols, and is not applicable here.

There are many different characterizations of PTIME such as Leivant's [128], Immerman's [115], Cobham's [38] and Bellantoni-Cook characterization [22]. Leivant's characterization says that PTIME functions are exactly those that are provable in a logic called $L_2(QF^+)$. Immerman's logic is a fixpoint logic with least fixpoint operator and $\leq$ which works on structures with abstract domain elements. The two other characterizations of Cobham's and Bellantoni-Cook have the property of characterizing PTIME as a set of functions working on numbers and so better suited for our purpose of characterizing search problems over arithmetical structures. Also, the safe recursion and safe composition operators in the Cook-Bellantoni characterization give us a more natural way of guaranteeing that the result of the simulation we need in our proof falls within some bounds. Therefore, we choose the Bellantoni-Cook characterization [22] over Cobham's as the basis of our proof.

Built-in arithmetic is implemented in many modelling languages, e.g. the MX-based IDP system [209] and LPARSE [182]. However, as we showed, such languages have limited expressiveness in the presence of arithmetic constraints. For example, we showed that the two problems of integer factorization and quadratic residues are not expressible in ASP and IDP systems using their built-in arithmetic. Also, in many cases, allowing arithmetic constraints without careful restrictions provides the language with very high expressiveness, as is shown for ESSENCE [151].

3.8 Conclusion

In modelling languages, one is frequently faced with the problem of having a framework to support both natural specifications of problems, and reasoning about those problems. In this chapter, we took our measure of naturality to be being able to use "built-in" arithmetic, and our measure of reasoning to be being in NP. We showed some examples of problems of practical importance and proved that several existing modelling languages cannot express these problems naturally (using their built-in arithmetic and not by encoding numbers using abstract domain elements). We also

presented a solution to this problem and we proved that embedded (in $\mathcal{N}^c$) MX for PBINT captures exactly NP. A consequence of this result is that our fragment of logic can represent all arithmetical problems in NP naturally. We supported our claim by giving natural axiomatizations for problems in NP that could not be naturally axiomatized in existing modelling languages. This result guarantees universality of our logic for this complexity class and also settles our reasoning abilities by showing that all PBINT axiomatizations can be efficiently (in polytime) grounded to any state of the art solver of NP problems. Our work is a significant step forward from the previous proposal since it overcomes a number of limitations.

As we showed in PBINT-Spec, logic PBINT is natural because it is essentially FO logic, where guards can be made "invisible" through "hiding" them in a type system. Solving for PBINT (also PBINT-Spec) can be achieved through grounding to SAT, a work which is being performed in our group, but falls outside of the topic of this chapter.

In summary, our work has pointed out some of the limitations of existing modelling languages and provided a solution to these limitations. Our work has shown a new application of descriptive complexity and metafinite model theory, and contributed to those areas by improving a previous result of capturing NP for arithmetical structures.

Future possible research directions include (a) to analyze other existing languages (such as DLV [49]) in connection with the logical fragments we defined; (b) to design other logics with useful background structures, and, (c) to analyze existing modelling languages with respect to new useful background structures.

Chapter 4

Modular Model Expansion

4.1 Motivation

Declarative programming is a branch of programming that is built on the idea of describing what we need to do instead of how we need to do it. Traditionally, declarative programming has been considered as an alternative to imperative or functional programming and it has mainly been used in the form of query languages for databases. However, the close relation between declarative programming paradigms, fragments of logic, and, the ability to interpret logical statements as specification of computation, has resulted in its wide adoption by many sub-communities of computing science as their paradigm of choice for (free of side-effect) manipulation of knowledge. One of the main such communities that has adopted declarative programming is the community of artificial intelligence (AI) researchers.

There are many different declarative languages that are used in AI applications. Examples of such languages include (but are not limited to): Integer Linear Programs (ILP) or Mixed Linear Programs (MLP), Answer Set Programs (ASP), Satisfiability (SAT), Satisfiability Modulo Theories (SMT), Constraint Programming (CP), etc. All of these languages have proved to be extremely successful in their domains of applications. For example, ILP has been extensively used to solve instances of vehicle routing problems. Also, CP has proved to be very efficient when dealing with scheduling problems while ASP is extremely good at solving different types of planning problems and reasoning with exceptions. Similarly, SAT is used widely for planning and solving other combinatorial problems such as automatic hardware design and verification. However, up until now, all these languages have been mostly focused on their own local knowledge: a trend that is changing slowly but that is also gaining pace due to recent advancements in the development of cloud-based

applications.

With the development of the cloud and various mobile applications, a possibly large number of autonomous, heterogeneous systems can collaborate to solve certain tasks. These agents may be communicating individuals, large businesses or corporations, or software systems. The tasks they may need to solve collaboratively can be computationally complex tasks such as optimizing the schedule of flights in a united group of airlines or computationally feasible but conceptually complex tasks such as planning an entertaining trip to India. While these two problems are of different complexities, they share many similarities on the representational fronts: *in order to solve them, one may need to access several databases with (perhaps) disjoint vocabularies and there may need to be an interaction between agents that represent entities with differing interests.*

Therefore, if tasks such as those above are to be represented in a declarative fashion, there should be a way to deal with their three most important properties: (1) their interactive nature (agents), (2) their computational complexity, and, (3) their conceptual complexity. In the past, the declarative programming community has intensively focused on handling the computational complexity of problems that occur in the real world. As a result, we believe that handling computational complexity is a less of a concern these days. One evidence to justify this claim is the existence of many different efficient solvers (such as ILP solvers, SAT solvers and ASP solvers) that easily deal with huge instances of computationally hard problems. While continuing the research on these solvers are necessary and fruitful to the whole community, we believe that, nowadays, the main challenge in declarative representation of hard problems is their *conceptual complexity*, i.e., there exist many common problems that are out of reach not because of their computational complexity but due to the complexity of representing them.

In this chapter, we try to tackle this new challenge of conceptual complexity using the principles of modular design. Modular design has been the de facto practice in engineering for many years. Software engineering has not been an exception to this rule either. Ever since the introduction of commercial software, modularity has been among the most agreed-upon guidelines in programming. However, in declarative programming, modularity is a relatively recent concern because, up until very recently, declarative programming was able to rely on a consistent and homogeneous setting of localized knowledge bases. However, knowledge bases are becoming global at a phenomenal pace and, thus, there is an ever-growing need to represent and reason about problems that need access to multiple, interacting and distributed knowledge bases which can also have heterogeneous semantics. In this chapter, we first proposes the foundations of declarative representation of modular systems. Then, we show how collaborative solving among autonomous systems and users can happen in a

modular fashion. Our eventual goal is to support distributed declarative programming that allows each peer/user to maintain better control of their data, and leverages the available computational resources.

Before continuing to the main content of this chapter, we want to briefly look at some different kinds of problems that need to be expressed in a modular way.

Example 4.1 (Modeling in Engineering) *For an engineer, a system is a collection of interacting modules. Thus, a model of a system is an interconnected set of modules. Such models help engineers to study various properties of systems such as their steady and/or transient states. While different fields in engineering use completely different models, they share an important property:* while using not very complex components, they are able to represent highly complex systems. *Therefore, the hard task is not to specify the behavior of components but to specify the behavior of the system. The focus of our work in this chapter is to build a framework that allows to combine primitive modules (that represent individual components) in complicated ways (to represent the whole system). This method allows an engineer to harness the power of engineering models in studying the properties of the final product while, simultaneously, restraining the conceptual complexity of the system.*

Example 4.2 (Rapid Software Prototyping) *In the previous example, we discussed how modular systems can be used to study engineering design models. Software engineering is no exception to this rule. Moreover, software engineering has the added benefit that, unlike other engineering fields, the product is a software. Therefore, the process of modeling software can be tightly coupled with the process of producing the final product. Rapid software prototyping uses such a philosophy to develop prototypes using abstract components that will be later refined to the actual software. Many software development methodologies (e.g., "Evolutionary Rapid Prototyping" [40, 47], "Executable UML" [144] and "Shlaer-Mellor method" [175, 144]) develop software in this way. They (1) use high-level and abstract modular representations of a system rather than detailed low-level and system-specific descriptions, and, (2) try to model final product's behaviour in the most accurate way possible. Thus, we believe that our work in this chapter can have immediate effect on the practice of rapid software prototyping through enabling the use of declarative programming languages in the process of industrial software production.*

Example 4.3 (Business Process Planning) *Another immediate application of current work is in business process planning. In general, such planning consists of finding ways to use contractors to perform a set S of services under some restrictions R. Here, S is the set of services offered by the*

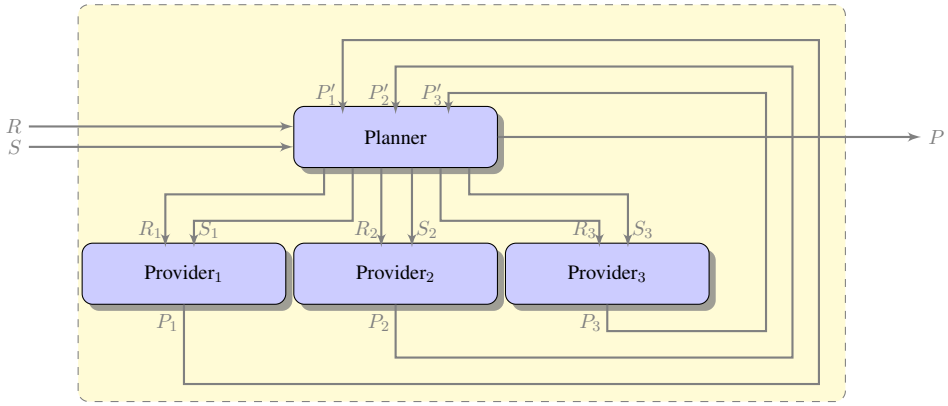

Figure 4.1: Business Process Planner.

main business and might be different from all the services provided by the contractors of the main business. For example, the service that a wedding planner provides is different from all the services provided by its many contractors such as florists, caterers and venue providers. Figure 4.1 shows an example of a business process planner that takes services S and restrictions R and splits them between service providers. In response, the planner generates plan P by combining partial plans P_i generated by providers. This example is also important because it justifies one of our main objectives in the current work, i.e., language-independence. In this example, language-independence is needed so that reasoning is still possible in the presence of modules that are not programmed declaratively, for example those modules that use legacy software, or, even those that represent a human agent. Hence, as we deem it essential to include all such cases, we believe that our modular system should take a completely language-independent approach towards solving.

The examples above demonstrate why it is a necessity for declarative programming community to enable the user to combine different systems and logics. Motivated by such a need, our goal in this chapter is to *introduce and investigate a modular approach towards specifying and solving complex tasks so that different and possibly heterogeneous parts can work together*. We are interested in finding a method that obtains solutions through orchestrating a collaborative effort between primitive modules.

Our first challenge is to develop an abstract formalism to represent modular declarative programs. This formalism should allow building systems from separate blocks on abstract model-theoretic level. Moreover, our modular framework should allow independent agents/systems to collaborate with each other and to have control over the maintenance of their own data (thus, allowing

both security and privacy from the point of view of the collaborating agents).

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.

This chapter continues the research program started by Mitchell and Ternovska [149] to develop the foundations of solving computationally hard search problems in declarative programming languages. They formalized search problems as the logical task of *model expansion (MX)*, i.e., to expand a structure with interpretations for new predicate/function symbols. This chapter extends and elaborates Tasharrofi and Ternovska [187] that was presented at the 2011 symposium on Frontiers of Combining Systems (FroCoS'11). In this chapter, we take model expansion beyond a particular language and introduce modular systems. Our modules can be viewed both model-theoretically and operationally. The model-theoretic view sees modules as sets (or classes) of structures, while the operational view sees them as operators on the underlying structure. There, modules are combined in an abstract algebraic form that provides for language-independence. Moreover, the loop (or feedback) operation of the algebra allows modelling of cyclically dependent modules and boosts the expressive power of the framework.

In addition to introducing modular system framework, we also describe a method to find *solutions* to modular systems. Here, a solution is a structure that is accepted by the modular system (as a function of individual modules). Thus, the goal here is to devise a way to find structures in a 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 [156].

Our second challenge in this work is to devise appropriate mathematical abstractions for the task 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 solve an extension of SAT with a theory, Constraint Answer Set solvers solve combinations of ASP and CP language, etc. In this chapter, we *design an algorithmic schema that, when initialized with a modular system M, takes a structure $\mathcal{A}$ and searches for expansions of $\mathcal{A}$ in M*. Note that we neither hope to compete with existing systems, nor hope to replace them, but to shed light on the general principles of solving complex tasks that are specified in a modular way. The result of our work is a general system that is not restricted to a particular language or a particular combination of languages. Our approach towards solving modular systems is to orchestrate a co-operative efforts between native solvers of all modules involved.

Contributions

Modular System Framework.

We introduce a framework to combine modules in a language-independent manner. Our framework allows for a range of operations including (serial/parallel) composition of modules, non-deterministic choice between modules, feedback, etc. that are common in system analyses. We define an algebraic language for describing modular systems and provide two natural semantics for this language: (a) a model-theoretic semantics that views modules as sets of structures, and, (b) an operational semantics that views modules as mappings between structures and computes fixpoints of those operators. We prove that these two semantics coincide and, thus, give complementary views on modular systems.

Complexity and Expressiveness.

We study the complexity and expressiveness of our modular system framework and some of its fragments (i.e., when certain operations are allowed/disallowed). We show that both our feedback operator and our non-deterministic choice operator hugely increase the expressiveness of our modular system framework.

Solving Modular Systems.

We introduce an algorithmic schema that, when initialized with a modular system, works as a model expansion solver for that modular system, i.e, given an input to that modular system, we can find one

of the outputs that correspond to the given input (if such an output exists) or report unsatisfiability (if such an output does not exist). We prove the correctness of our algorithm and study its computational complexity. Moreover, in order to design such an algorithm, we define several abstract concepts that, if satisfied by modules of a modular system, can tremendously speed up the process of solving instances of that modular system. Moreover, these concepts provide guidelines on how to augment a module in a modular system so that collaborative solving is achieved.

Case Study of Three (Mixed) Solvers.

We investigate three state-of-the-art types of solvers that use a mixed solving approach (i.e., mix two or more methods to achieve better solving performance): (a) DPLL(T) solvers that combines DPLL[1] algorithm with the solver for theory T, (b) mixed integer program solvers that combine branching algorithms and cut generation algorithms with algorithms for solving linear programs (such as the Simplex algorithm [44]) to obtain efficient algorithms for solving mixed linear programs, and, (c) recent efforts to combine Answer Set Programming with Constraint Programming. For each of the three types of solvers, we give a modular system with the "right" modules so that when our algorithmic schema is initialized with that "right" modular system, it acts similar to the native solver of that system. That is, if our algorithmic schema is initialized with the modular system that represents mixed linear solvers, it acts similar to mixed linear solvers and, when initialized with modular system that represents a combination of ASP and CP, it acts similar to the native ASP+CP solvers. We use these observations to argue for the efficiency of our algorithm in solving modular systems in general (even when modules are specified in a not so well-studied language).

Approximation of Solutions to Modular Systems.

We introduce two classes of modular systems based on simple properties that modules of a modular system may provide. For the first class of modular systems, we introduce a new procedure to obtain (in polynomial time) at least one of the solutions of modular systems in that class. Moreover, the solution we obtain for those modules approximates (in a sense that will be defined later) all other solutions of that modular system as well. For the second class of modular systems, we give another polynomial time procedure that obtains a non-trivial lower bound and upper bound (thus, an approximation) for all solutions of a modular system. We further modify both of these procedures so that they can be used for obtaining stronger bounds within our algorithmic schema for solving

[1] Davis–Putnam–Logemann–Loveland [157].

modular systems. We then incorporate these approximation procedures in our algorithmic schema and obtain a more intelligent way of combining solving mechanisms.

4.2 Background

4.2.1 Model Expansion

Recall that the authors of [149] formalized combinatorial search problems as the task of *model expansion (MX)*, i.e., the logical task of expanding a given (mathematical) structure with new relations. In the model expansion task, we expand a σ-structure $\mathcal{A}$ with relations and functions to interpret new vocabulary ε, obtaining a structure $\mathcal{B}$ that satisfies our axiomatization ϕ in a logic $\mathcal{L}$. Also, recall that σ, the vocabulary of $\mathcal{A}$, is called the *instance* vocabulary, and ε is called the *expansion* vocabulary.

Example 4.4 (Business Process Planner as Model Expansion) *In Figure 4.1, both the planner box and the provider boxes can be viewed as model expansion tasks. For example, the box labeled with "Provider$_1$" can be abstractly viewed as an MX task with instance vocabulary $\sigma = \{S_1, R_1\}$ and expansion vocabulary $\varepsilon = \{P_1\}$. The task is: given some services S_1 and some restrictions R_1, find a plan P_1 to deliver services in S_1 such that all restrictions in R_1 are satisfied.*

Moreover, in Figure 4.1, the bigger box with dashed borders can also be viewed as an MX task with instance vocabulary $\sigma' = \{S, R\}$ and expansion vocabulary $\varepsilon' = \{P\}$. This task is a compound MX task whose result depends on the internal work of all the providers and the planner.

Given a specification, we can talk about a set of $\sigma \cup \varepsilon$-structures which satisfy the specification. Alternatively, recall that we can also talk about a given set of $\sigma \cup \varepsilon$-structures as an MX-task, without mentioning a particular specification the structures satisfy. This abstract view makes our study of modularity language-independent.

4.2.2 Partial Structures and Extensions

Recall that a structure is a domain together with an interpretation of the associated vocabulary that consists of a finite set of predicate and function symbols (including null-ary functions or constants). A partial structure extends the concept of a structure by allowing some interpretations (e.g., the interpretation $R^{\mathcal{B}}$ of n-ary predicate symbol R in a partial structure $\mathcal{B}$) may be partially specified (e.g., for some tuple $\bar{a} \in [dom(\mathcal{B})]^n$, we may not know whether $\bar{a} \in R^{\mathcal{B}}$ or $\bar{a} \notin R^{\mathcal{B}}$). Partial structures generalize structures in the sense that a structure is a particular partial structure with fully

understood interpretations. Partial structures usually encode our partial knowledge about a subject and arise naturally in computational problems. We will see several natural uses for partial structures in this chapter (e.g., in Section 4.5).

Definition 4.1 (Partial Structure) *$\mathcal{B}$ 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 $R^+ \cup R^- \subsetneq [dom(\mathcal{B})]^n$.*

We say that τ_p is the partial vocabulary *of $\mathcal{B}$. We say $\mathcal{B}$ is* total *if $\tau_p = \emptyset$.*

Note that, as expected, a total structure is just a τ_p-partial structure according to definition 4.1 with an empty vocabulary τ_p.

Example 4.5 *Consider a structure $\mathcal{B}$ over 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 $R^{-^{\mathcal{B}}} = \{\langle 1\rangle\}$.*

Partial structures can encode the presence or lack of information about the truth of a proposition at certain points. Therefore, it is natural to compare two partial structures with respect to their amount of information (or lack thereof). In the following, we introduce extensions, positive extensions and negative extensions that define natural partial orderings on partial structures.

Definition 4.2 (Positive/Negative Information) *Let $\mathcal{B}$ be a τ_p-partial structure over vocabulary τ. Then, by $\mathcal{B}^+$, we denote the positive information in $\mathcal{B}$, i.e., information about the presence of tuples in interpretations given by $\mathcal{B}$:*

$$\mathcal{B}^+ := \{R(\bar{a}) \mid \textit{either } [R \in (\tau \setminus \tau_p) \textit{ and } \bar{a} \in R^{\mathcal{B}}] \textit{ or } [R \in \tau_p \textit{ and } \bar{a} \in R^{+^{\mathcal{B}}}]\}.$$

Similarly, $\mathcal{B}^-$ denotes the negative information in $\mathcal{B}$, i.e., the information about the absence of tuples in interpretations:

$$\mathcal{B}^- := \{R(\bar{a}) \mid \textit{either } [R \in (\tau \setminus \tau_p) \textit{ and } \bar{a} \notin R^{\mathcal{B}}] \textit{ or } [R \in \tau_p \textit{ and } \bar{a} \in R^{-^{\mathcal{B}}}]\}.$$

Definition 4.3 (Extension, Positive Extension, Negative Extension) *Let $\mathcal{B}$ and $\mathcal{B}'$ be two partial structures over the same vocabulary and the same domain. We say that $\mathcal{B}'$* positively extends *$\mathcal{B}$, denoted by $\mathcal{B} \sqsubseteq_P \mathcal{B}'$, if $\mathcal{B}^+ \subseteq \mathcal{B}'^+$, i.e., $\mathcal{B}'$ has at least as much positive information as does $\mathcal{B}$. Similarly, we say that $\mathcal{B}'$* negatively extends *$\mathcal{B}$, denoted by $\mathcal{B} \sqsubseteq_N \mathcal{B}'$ if $\mathcal{B}^- \subseteq \mathcal{B}'^-$. We also say that $\mathcal{B}'$* extends *$\mathcal{B}$, denoted by $\mathcal{B} \sqsubseteq \mathcal{B}'$ if both $\mathcal{B} \sqsubseteq_P \mathcal{B}'$ and $\mathcal{B} \sqsubseteq_N \mathcal{B}'$.*

Note that, according to Definition 4.3, if $\mathcal{B}$ and $\mathcal{B}'$ are total structures, i.e., all predicate symbols are fully interpreted, then $\mathcal{B}$ extends $\mathcal{B}'$ if and only if $\mathcal{B} = \mathcal{B}'$. This is due to the fact that extension respects both the positive information and the negative information that is contained in a partial structure. Thus, as two different total structures should necessarily contradict each other on some piece of information, neither of them can carry "more information" than the other one and, hence, neither of them extends the other one. On the other hand, positive and negative extensions define a more relaxed notion of extension that will be useful when we want to talk about monotonicity or anti-monotonicity of operators on structures.

We sometimes abuse the notation and, for (possibly partial) interpretations S_1 and S_2 of a single vocabulary symbol S in two structures over the same domain, write $S_1 \sqsubseteq S_2$ (resp. $S_1 \sqsubseteq_P S_2$ or $S_1 \sqsubseteq_N S_2$ to say that S_2 extends (resp. positively extends or negatively extends) S_1. We may also write $S_1 \sqcap S_2$ (resp. $S_1 \sqcap_P S_2$ or $S_1 \sqcap_N S_2$) to denote the information that is shared in interpretations S_1 and S_2 (resp. the positive or negative shared information) for predicate symbol S. We may use $\sqcup$, $\sqcup_P$ and $\sqcup_N$ similarly. We also use $\sqsubset$ and $\sqsupset$ (resp. $\sqsubset_P$, $\sqsubset_N$, $\sqsupset_P$ and $\sqsupset_N$) to denote proper extension, i.e., similar to $\sqsubseteq$ and $\sqsupseteq$ but without equality.

Finally, also let us define concatenation of two structures as follows.

Definition 4.4 (Concatenation of Structures ($\mathcal{B}_1||\mathcal{B}_2$)) *Let $\mathcal{B}_1$ and $\mathcal{B}_2$ be two structures over distinct vocabularies such that $dom(\mathcal{B}_1) = dom(\mathcal{B}_2)$. Then, by $\mathcal{B}_1||\mathcal{B}_2$, we denote a structure $\mathcal{B}$ with the same domain as $\mathcal{B}_1$ and $\mathcal{B}_2$ but over the vocabulary of both $\mathcal{B}_1$ and $\mathcal{B}_2$, i.e., $vocab(\mathcal{B}) = vocab(\mathcal{B}_1) \cup vocab(\mathcal{B}_2)$. Moreover, $\mathcal{B}$ is defined so that it takes the interpretation of its vocabulary symbols from the corresponding structure, i.e., for all symbols $R \in vocab(\mathcal{B})$, we have that $R^{\mathcal{B}} = R^{\mathcal{B}_1}$ if $R \in vocab(\mathcal{B}_1)$ and $R^{\mathcal{B}} = R^{\mathcal{B}_2}$ if $R \in vocab(\mathcal{B}_2)$.*

4.3 Modular Systems

In this section, we present the concept of a modular system, which we introduced in our earlier conference paper [187]. We also extend and refine our previous notion of a modular system as an operator. The initial development was motivated by [120], however our framework offers two equivalent semantics based on model-theoretic and operational views (none of them present in [120]). Also, as we explain in this section, our framework extends that earlier work in several significant ways and, in particular, by adding an expressive feedback operator. In the following, we first formally define an algebraic language for modular systems and then two (equivalent) semantics for this language. The first semantics is model-theoretical and recursively defines the set of structures that a modular system abstractly represents. The second semantics is called the fixpoint semantics and associates a non-deterministic operator to each modular system. We prove that these two semantics coincide (i.e., the second semantics associates an operator to a modular system whose fixpoints are closely tied to the structures that are associated to the modular system by the first semantics). This way, we have two distinct methods to study the properties of modular systems, and, as we shall see, both these methods are advantageous in certain situations.

4.3.1 The Algebra of Modular Systems

As in [187], *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. Intuitively, the projection operator on M defines a modular system that acts as M internally but is viewed differently from outside by hiding some vocabulary symbols that are meaningful only internally.

(2) Composition($M_1 \rhd M_2$) which connects outputs of M_1 to inputs of M_2. As its name suggests, the composition operator is intended to take two modular systems and defines a multi-step operation by serially composing M_1 and M_2.

(3) Union($M_1 \cup M_2$) which, intuitively, models the case when we have two alternatives to do a task (that we can choose from).

(4) Feedback($M[R = S]$) which connects output S of M to its inputs R. As the name suggests, the feedback operator models systems with feedback.

The algebra of modular systems is defined recursively starting from primitive modules:

Definition 4.5 (Primitive Module [187]) *A* primitive module M *is a model expansion task (or, equivalently, a search problem) with instance vocabulary σ and expansion vocabulary ε (again, equivalently, with input vocabulary σ and output vocabulary ε). Associated with each primitive module M, there is a decision procedure D_M such that, given a $(\sigma_M \cup \varepsilon_M)$-structure $\mathcal{B}$, D_M accepts $\mathcal{B}$ if and only if M describes a task of model expansion that accepts $\mathcal{B}$ as a valid expansion of $\mathcal{B}|_\sigma$.*

Remark 4.1 (Representation of Primitive Modules) *A primitive module as described in Definition 4.5 does not have to conform to a specific representation. For example, formula ϕ of Example 2.1 describes the model expansion task for the problem of Graph 3-coloring. Thus, ϕ can be the representation of a module M_C with instance vocabulary $\{E\}$ and expansion vocabulary $\{R, G, B\}$. However, since each primitive module M according to Definition 4.5 is associated with a decision procedure D_M, there exists a set-theoretical and canonical representation for all primitive modules: a set (class) of $(\sigma_M \cup \varepsilon_M)$-structures, i.e.,* $\mathrm{Rep}(M) := \{\mathcal{B} \mid \mathcal{B}$ *is a $(\sigma \cup \varepsilon)$-structure and* $D_M(\mathcal{B}) = 1\}$. *As a result, we sometimes use M as a set to denote* $\mathrm{Rep}(M)$. *Note that this canonical representation of M is just one of its representation and does not determine how M is represented in reality. In fact, in most cases, M is not represented in this way and has a finite representation like in Example 2.1.*

Before recursively defining our algebraic language, we have to define composable and independent modules:

Definition 4.6 (Composable, Independent [120]) *Modules M_1 and M_2 are* composable *if $\varepsilon_{M_1} \cap \varepsilon_{M_2} = \emptyset$ (no output interference). Module M_2 is* independent *from M_1 if $\sigma_{M_2} \cap \varepsilon_{M_1} = \emptyset$ (no cyclic module dependencies).*

Definition 4.7 (Well-Formed Modular Systems ($\mathrm{MS}(\sigma, \varepsilon)$)) *The* set of all well-formed modular systems $\mathrm{MS}(\sigma, \varepsilon)$ *for a given input, σ, and output, ε, vocabularies is defined as follows.*

Base Case, Primitive Modules: *If M is a primitive module with instance (input) vocabulary σ and expansion (output) vocabulary ε, then $M \in \mathrm{MS}(\sigma, \varepsilon)$.*

Projection *If $M \in \mathrm{MS}(\sigma, \varepsilon)$ and $\tau \subseteq \sigma \cup \varepsilon$, then $\pi_\tau(M) \in \mathrm{MS}(\sigma \cap \tau, \varepsilon \cap \tau)$.*

Sequential Composition: *If $M \in \mathrm{MS}(\sigma, \varepsilon)$, $M' \in \mathrm{MS}(\sigma', \varepsilon')$, M is composable (no output interference) with M', and M is independent from M' (no cyclic dependencies) then $(M \rhd M') \in \mathrm{MS}(\sigma \cup (\sigma' \setminus \varepsilon), \varepsilon \cup \varepsilon')$.*

Union: *If* $M \in \mathrm{MS}(\sigma, \varepsilon)$, $M' \in \mathrm{MS}(\sigma', \varepsilon')$, M *is independent from* M', *and* M' *is also independent from* M *then* $(M \cup M') \in \mathrm{MS}(\sigma \cup \sigma', \varepsilon \cup \varepsilon')$.

Feedback: *If* $M \in \mathrm{MS}(\sigma, \varepsilon)$, $R \in \sigma$, $S \in \varepsilon$, *and* R *and* S *are symbols of the same type and arity, then* $M[R = S] \in \mathrm{MS}(\sigma \setminus \{R\}, \varepsilon \cup \{R\})$.

Nothing else is in the set $\mathrm{MS}(\sigma, \varepsilon)$.

Further operators for combining modules can be defined as combinations of basic operators above. For instance, [120] introduced $M_1 \blacktriangleright M_2$ operator (composition operator combined with projection) as $\pi_{\sigma_{M_1} \cup \varepsilon_{M_2}}(M_1 \rhd M_2)$, i.e., serial composition of M_1 and M_2 with the intermediate results (generated by M_1) forgotten. Also, for $M_1 \in \mathrm{MS}(\sigma_1, \varepsilon_1)$ and $M_2 \in \mathrm{MS}(\sigma_2, \varepsilon_2)$, $M_1 \cap M_2$ is defined to be equivalent to $M_1 \rhd M_2$ (or $M_2 \rhd M_1$) when $\sigma_1 \cap \varepsilon_2 = \sigma_2 \cap \varepsilon_1 = \varepsilon_1 \cap \varepsilon_2 = \emptyset$, i.e., $M_1 \cap M_2$ denotes the composition of two mutually independent components in a system.

Example 4.6 (BPP as a Modular System) *Let us illustrate the operations of modular systems by giving an algebraic specification of the modular system in Example 4.3.*

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

Considering Figure 4.1, symbol BPP *refers to the whole modular system denoted by the box with dotted borders. Also, as* R, S *and* P *are the only vocabulary symbols important outside* BPP, *all other symbols in Formula* (4.1) *are projected out. Moreover, primitive modules* $Provider_1$, $Provider_2$ *and* $Provider_3$ *represent three independent providers. Thus, they are connected using operator* $\cap$ *(composition of independent modular systems). Furthermore, there are three feedbacks from* P_i *(*$i \in \{1, 2, 3\}$*) to* P_i' *that return the partial plans generated by providers to the primitive module* $Planner$.

The description of a modular system (as in Definition 4.7) gives a formula representing a system. Thus, it is convenient to define *subsystems* of a modular system M as sub-formulas of the formula that represents M. Clearly, each subsystem of a modular system is a modular system itself.

4.3.2 Model-theoretic Semantics for Modular Systems

In Section 4.3.1, we introduced the syntax of our algebraic language. The basic elements in Definition 4.7 are primitive modules that are sets of structures (based on Definition 4.5). Our first semantics

for modular systems extends this approach by defining the set of structures that are represented by a complex modular system. Each such structure is called a *model* of that modular system:

Definition 4.8 (Models of a Modular System) *Let $M \in \mathrm{MS}(\sigma, \varepsilon)$ be a modular system and $\mathcal{B}$ be a $(\sigma \cup \varepsilon)$-structure. The following gives a recursive definition of $\mathcal{B}$ being a* model *of M:*

Base Case, Primitive Modules: *$\mathcal{B}$ is a model of M if and only if $\mathcal{B} \in M$.*

Projection: *$\mathcal{B}$ is a model of $M := \pi_{(\sigma \cup \varepsilon)}(M')$ (with $M' \in \mathrm{MS}(\sigma', \varepsilon')$) if an only if a $(\sigma' \cup \varepsilon')$-structure $\mathcal{B}'$ exists such that $\mathcal{B}'$ is a model of M' and $\mathcal{B}'$ expands $\mathcal{B}$.*

Composition. *$\mathcal{B}$ is a model of $M := M_1 \rhd M_2$ (with $M_1 \in \mathrm{MS}(\sigma_1, \varepsilon_1)$ and $M_2 \in \mathrm{MS}(\sigma_2, \varepsilon_2)$) if and only if $\mathcal{B}|_{(\sigma_1 \cup \varepsilon_1)}$ is a model of M_1 and $\mathcal{B}|_{(\sigma_2 \cup \varepsilon_2)}$ is a model of M_2.*

Union. *$\mathcal{B}$ is a model of $M := M_1 \cup M_2$ (with $M_1 \in \mathrm{MS}(\sigma_1, \varepsilon_1)$ and $M_2 \in \mathrm{MS}(\sigma_2, \varepsilon_2)$) if and only if either $\mathcal{B}|_{(\sigma_1 \cup \varepsilon_1)}$ is a model of M_1, or $\mathcal{B}|_{(\sigma_2 \cup \varepsilon_2)}$ is a model of M_2.*

Feedback. *$\mathcal{B}$ is a model of $M := M'[R = S]$ (with $M' \in \mathrm{MS}(\sigma', \varepsilon')$) if and only if $R^{\mathcal{B}} = S^{\mathcal{B}}$ and $\mathcal{B}$ is model of M'.*

In this chapter, we are mainly interested in the task of model expansion for modular systems that is defined as below:

Definition 4.9 (Model Expansion for Modular Systems) *Let $M \in \mathrm{MS}(\sigma, \varepsilon)$ be a modular system. The task of model expansion for M takes a σ-structure $\mathcal{A}$ and finds (or reports that none exists) a $(\sigma \cup \varepsilon)$-structure $\mathcal{B}$ that expands $\mathcal{A}$ and is a model of M. Such a structure $\mathcal{B}$ is a* solution of M for input $\mathcal{A}$.

Comparison with [120] The framework [120] is based on a set of variables $\mathcal{X}$ with each $x \in \mathcal{X}$ having a domain $D(x)$. An *assignment* over a subset of variables $X \subseteq \mathcal{X}$ is a function $\sigma : X \to \cup_{x \in X} D(x)$, which maps variables in X to values in their domains, i.e., $\sigma(x) \in D(x)$ for all $x \in X$. A constraint C over a set of variables X is characterized by a set C of assignments over X, called the *satisfying assignments*. A primitive module in terms of [120] is a constraint plus a signature for its input and outputs, i.e., two disjoint sets $I, O \subseteq X$ of variables. Modular systems in [120] are combined using operators of composition and projection, i.e., there is no union or feedback operators. In this chapter, the variables of [120] are represented by the vocabulary symbols in $\sigma \cup \varepsilon$. Moreover, instead of assigning values, we use structures that assign interpretations to vocabulary symbols. Therefore, unlike [120] that is concerned with the task of satisfiability checking (i.e., it tries to find an assignment that satisfies all the constraint), we are concerned with the task of model expansion that is suitable for modeling the whole system in a modular fashion (and not just one input of the system).

4.3.3 Fixpoint Semantics for Modular Systems

In Section 6.1, we mentioned that one of the motivating factors behind the development of modular systems was their broad applicability to model the steady state analysis and the transient state analysis of complex systems. In Section 4.3.2, we defined the intended models of a modular system. However, in order to analyze steady and/or transient states of a system in terms of its modular representation, we should first have a natural definition of the state of a modular system as well as the necessary means to view a modular system as an operator on this state. In this section, we introduce a new semantics for modular systems called *fixpoint semantics*. As we see in this section, while our fixpoint semantics is very closely related to our model-theoretical semantics, it views modular systems in an intrinsically different way as operators on some states.

The the operational view that we introduce in this section enables us to obtain interesting results about our modular systems such as approximability of a sub-class of modular systems. Moreover, the concept of time naturally arises from our operational view towards modular systems. We will briefly review this concept in the end of this section and discuss the close relation between our concept of time in modular systems and the transient state analysis in an engineering system. The full treatment of this concept would be the subject of a future research.

Before defining the fixpoint semantics of modular systems, we first define the concept of a state of a modular system. Then, to each modular system, we associate an operator that operators on a state of a modular system. Next, as we will see later on, our fixpoint semantics would be simply defined as the fixpoints of that operator.

Definition 4.10 (State of a Modular Systems) *Let $M \in \mathrm{MS}(\sigma, \varepsilon)$ be a modular system. Then a* state *of the modular system M is simply a τ-structure such that $(\sigma \cup \varepsilon) \subseteq \tau$.*

Now, let us associate with a modular system an operator on states of that modular system as follows:

Definition 4.11 (Operational View on Modular Systems) *Let $M \in \mathrm{MS}(\sigma, \varepsilon)$ be a modular system and let τ-structure $\mathcal{B}_1$ be a state of M. Now, we say that a τ-structure $\mathcal{B}_2$* is a result of applying (non-deterministic) operator M on $\mathcal{B}_1$*, denoted as either $\mathcal{B}_1[\![M]\!]\mathcal{B}_2$ or $\mathcal{B}_2 \in [\![M]\!](\mathcal{B}_1)$ and defined as follows:*

Base Case, Primitive Modules: *$\mathcal{B}_1[\![M]\!]\mathcal{B}_2$ if and only if $\mathcal{B}_2|_{\sigma\cup\varepsilon} \in M$, and, $\mathcal{B}_2|_{(\tau\setminus\varepsilon)} = \mathcal{B}_1|_{(\tau\setminus\varepsilon)}$*[2]*,*

[2] *In particular, it means that $\mathcal{B}_1$ and $\mathcal{B}_2$ should have the same domain.*

Projection: *If* $M := \pi_{(\sigma\cup\varepsilon)}(M')$ *(with* $M' \in \mathrm{MS}(\sigma', \varepsilon')$*) then* $\mathcal{B}_1[\![M]\!]\mathcal{B}_2$ *if and only if* $(\sigma' \cup \varepsilon')$*-structures* $\mathcal{B}'_1$ *and* $\mathcal{B}'_2$ *exist such that*

1. $\mathcal{B}'_1|_{(\sigma\cup\varepsilon)} = \mathcal{B}_1|_{(\sigma\cup\varepsilon)}$ *(expanding* $\mathcal{B}_1$ *with interpretations of projected-out symbols obtains* $\mathcal{B}'_1$*),*
2. $\mathcal{B}'_1[\![M']\!]\mathcal{B}'_2$ *(applying* M' *on* $\mathcal{B}'_1$ *obtains* $\mathcal{B}'_2$*),*
3. $\mathcal{B}'_2|_{(\sigma\cup\varepsilon)} = \mathcal{B}_2|_{(\sigma\cup\varepsilon)}$ *(projecting* $\mathcal{B}'_2$ *on* $\sigma \cup \varepsilon$ *obtains* $\mathcal{B}_2$*), and,*
4. $\mathcal{B}_1|_{\tau\setminus(\sigma\cup\varepsilon)} = \mathcal{B}_2|_{\tau\setminus(\sigma\cup\varepsilon)}$ *(*M *can only affect its vocabulary).*

Composition: *If* $M := M_1 \rhd M_2$ *(with* $M_1 \in \mathrm{MS}(\sigma_1, \varepsilon_1)$ *and* $M_2 \in \mathrm{MS}(\sigma_2, \varepsilon_2)$*) then* $\mathcal{B}_1[\![M]\!]\mathcal{B}_2$ *if and only if* τ*-structure* $\mathcal{B}'$ *exists such that* $\mathcal{B}_1[\![M_1]\!]\mathcal{B}'$ *and* $\mathcal{B}'[\![M_2]\!]\mathcal{B}_2$*,*

Union. *If* $M := M_1 \cup M_2$ *(with* $M_1 \in \mathrm{MS}(\sigma_1, \varepsilon_1)$ *and* $M_2 \in \mathrm{MS}(\sigma_2, \varepsilon_2)$*) then* $\mathcal{B}_1[\![M]\!]\mathcal{B}_2$ *if and only if either* $\mathcal{B}_1[\![M_1]\!]\mathcal{B}_2$ *or* $\mathcal{B}_1[\![M_2]\!]\mathcal{B}_2$*,*

Feedback: *If* $M := M'[R = S]$ *(with* $M' \in \mathrm{MS}(\sigma', \varepsilon')$*) then* $\mathcal{B}_1[\![M]\!]\mathcal{B}_2$ *if and only if natural number* $n \geq 1$ *and* τ*-structures* $\mathcal{B}_1^0, \cdots, \mathcal{B}_1^n$ *and* $\mathcal{B}_2^0, \cdots, \mathcal{B}_2^{n-1}$ *exist such that:*

$$
\begin{aligned}
&\mathcal{B}_1^0 = \mathcal{B}_1, \mathcal{B}_1^n = \mathcal{B}_2,\\
&\mathcal{B}_1^i[\![M']\!]\mathcal{B}_2^i \text{ for all } i \text{ such that } 0 \leq i < n,\\
&\mathcal{B}_1^{i+1}|_{(\tau\setminus\{R\})} = \mathcal{B}_2^i|_{(\tau\setminus\{R\})} \text{ for all } i \text{ such that } 0 \leq i < n,\\
&R^{\mathcal{B}_1^{i+1}} = S^{\mathcal{B}_2^i} \text{ for all } i \text{ such that } 0 \leq i < n,\\
&R^{\mathcal{B}_1^n} = S^{\mathcal{B}_1^n}.
\end{aligned}
$$

Intuitively, the operator associated with a modular system M takes a structure $\mathcal{B}$ and generates $\mathcal{B}'$ by changing interpretations of the expansion vocabulary such that the result is a structure in M. This goal is attained through an operational means with a very natural meaning for each connective in our modular framework. For example, the meaning of connective $\rhd$ is a sequential composition of the two modular systems. Also, connective $\cup$ gives a non-deterministic choice on which modular system to execute and the feedback operator defines a loop. Among different cases of Definition 4.11, the two cases of projection and feedback are the most complex ones. Figures 4.2 and 4.3 illustrate the internal operations of these two operators.

Definition 4.11 gives an alternative semantics to a combined modular system. Unlike Definition 4.7 that treated modular systems as sets of structures, in Definition 4.11, modular systems are viewed as non-deterministic operators mapping structures to structures. Now that we have this operational view on modular systems, we can define the fixpoint semantics of a modular system simply as the fixpoint of their corresponding operator.

Figure 4.2: Module $M := \pi_\tau(M')$ operates by (a) expanding vocabulary of input $\mathcal{B}_1$, (b) applying M' to expanded input, and, (c) projecting the result of application of M'.

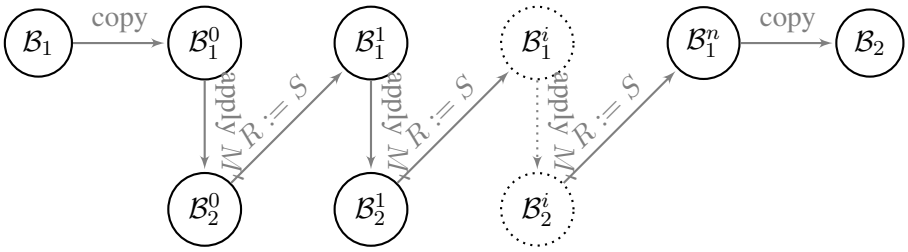

Figure 4.3: Module $M := M'[R = S]$ operates by repeatedly applying M' on the given structure and copying interpretation of S into interpretation of R until it reaches a fixpoint of M'.

Definition 4.12 (Fixpoint Semantics of Modular Systems) *Let $M \in \mathrm{MS}(\sigma, \varepsilon)$ be a modular system and $\mathcal{B}$ be a τ-structure (with $\tau \supseteq (\sigma \cup \varepsilon)$). We say that $\mathcal{B}$ is a* fixpoint *of M if and only if $\mathcal{B} \in [\![M]\!](\mathcal{B})$ (or, equivalently but with a different notation: $\mathcal{B}[\![M]\!]\mathcal{B}$).*

The following theorem shows that fixpoint semantics of modular systems coincides with the model-theoretic definition of a modular system:

Theorem 4.1 (Fixpoint Semantics = Model-theoretical Semantics) *Let M be a modular system. Then, for every τ-structure $\mathcal{B}$:*

$$\mathcal{B} \in [\![M]\!](\mathcal{B}) \iff \mathcal{B}|_{vocab(M)} \in M.$$

The most important consequence of Theorem 4.1 is that all the results obtained when modules are viewed as operators, remain valid when modules are viewed as sets of structures (and vice versa). Thus, in this chapter, we may use either of these semantics.

Remark 4.2 *Notice that both our model-theoretic semantics for modular system and our fixpoint semantics for modular systems are defined so that they do not put any finiteness restriction on the domains of structures. Thus, our modular system framework readily supports working on modules with infinite structures.*

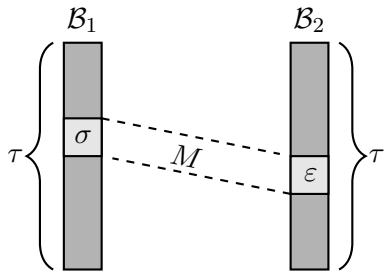

Figure 4.4: Module $M \in \mathrm{MS}(\sigma, \varepsilon)$ maps a τ-structure $\mathcal{B}_1$ (with $\tau \supseteq (\sigma \cup \varepsilon)$) to a τ-structure $\mathcal{B}_2$ by changing the interpretation of vocabulary symbols in ε according to the models of M (so that the σ part and the new ε part, together, form a model of M). Interpretation of all other symbols, including those in σ, stays the same.

The theorem that follows shows another important correspondence between our fixpoint semantics and our model-theoretical semantics.

Theorem 4.2 (Dual View on Modular Systems) *Let $M \in \mathrm{MS}(\sigma, \varepsilon)$ and also let $\mathcal{B}_1$ and $\mathcal{B}_2$ be two τ-structures such that $\mathcal{B}_1 [\![M]\!] \mathcal{B}_2$. Now, define $\mathcal{B}$ to be a $(\sigma \cup \varepsilon)$-structure such that: (1) $dom(\mathcal{B}) = dom(\mathcal{B}_1) = dom(\mathcal{B}_2)$, (2) $\mathcal{B}|_\sigma = \mathcal{B}_1|_\sigma$, and, (3) $\mathcal{B}|_\varepsilon = \mathcal{B}_2|_\varepsilon$. Then, $\mathcal{B} \in M$.*

The importance of Theorem 4.2 is to show how a modular system changes its state. This is also illustrated by Fiqure 4.4. Figure 4.4 shows that a modular system $M \in \mathrm{MS}(\sigma, \varepsilon)$ changes only the ε interpretations of the state and everything else (including σ) remains unchanged. This is similar to how frame axioms keep fluents that are not affected by actions unchanged in sequent calculus.

Structural Operational Semantics.

We have just introduced operational view on modular systems and the corresponding fixpoint semantics of modular systems. Now, we want to define the operational semantics of modular systems that is closely related to the operational view we developed above.

In this part, as in the operational view on modular systems, the vocabulary of structures include both the input and output vocabularies of M, i.e., $\sigma \cup \varepsilon \subseteq \tau$. Below, we give a structural operational semantics for the algebra of modular systems. The semantics is structural because, for example, the meaning of the sequential composition, $M_1 \rhd M_2$, is defined through the meaning of M_1 and the meaning of M_2. We start by defining the rules of structural operational semantics for modular systems.

Primitive modules.: If $M \in \mathrm{MS}(\sigma, \varepsilon)$ is a primitive module, then the only possible structural operational rule for M is as follows:

$$\frac{(M, \mathcal{B}_1) \longrightarrow \mathcal{B}_2}{true} \text{ if } \mathcal{B}_2|_{(\sigma \cup \varepsilon)} \in M \text{ and } \mathcal{B}_2|_{(\tau \setminus \varepsilon)} = \mathcal{B}_1|_{(\tau \setminus \varepsilon)}.$$

Projection.: The only possible structural operational rule for modular system $\pi_\nu(M)$ is as follows:

$$\frac{(\pi_\nu(M), \mathcal{B}_1) \longrightarrow \mathcal{B}_2}{(M, \mathcal{B}'_1) \longrightarrow \mathcal{B}'_2} \text{ if } \mathcal{B}'_1|_\nu = \mathcal{B}_1|_\nu \text{ and } \mathcal{B}'_2|_\nu = \mathcal{B}_2|_\nu.$$

Composition.: The only possible structural operational rule for modular system $M_1 \rhd M_2$ is as follows:

$$\frac{(M_1 \rhd M_2, \mathcal{B}_1) \longrightarrow \mathcal{B}_2}{(M_1, \mathcal{B}_1) \longrightarrow \mathcal{B}' \text{ and } (M_2, \mathcal{B}') \longrightarrow \mathcal{B}_2}.$$

Union.: For modular system $M_1 \cup M_2$, there exist two possible structural operational rules:

$$\frac{(M_1 \cup M_2, \mathcal{B}_1) \longrightarrow \mathcal{B}_2}{(M_1, \mathcal{B}_1) \longrightarrow \mathcal{B}_2}, \qquad \frac{(M_1 \cup M_2, \mathcal{B}_1) \longrightarrow \mathcal{B}_2}{(M_2, \mathcal{B}_1) \longrightarrow \mathcal{B}_2}.$$

Feedback.: For modular system $M[R = S]$, there are also two structural operational rules that follow:

$$\frac{(M[R = S], \mathcal{B}) \longrightarrow \mathcal{B}}{(M, \mathcal{B}) \longrightarrow \mathcal{B}}, \qquad \frac{(M[R = S], \mathcal{B}_1) \longrightarrow \mathcal{B}_2}{(M, \mathcal{B}_1) \longrightarrow \mathcal{B}' \text{ and } (M[R = S], \mathcal{B}'[R := S]) \longrightarrow \mathcal{B}_2},$$

where $\mathcal{B}'[R := S]$ is a τ-structure $\mathcal{B}''$ that agrees with $\mathcal{B}'$ on both its domain and the interpretations of all symbols in τ except R. Also, the interpretation of R in $\mathcal{B}''$ is a copy of the interpretation of S in $\mathcal{B}'$.

Definition 4.13 (Derivability) *For a well-defined modular system M, we say that $(M, \mathcal{B}_1) \longrightarrow \mathcal{B}_2$ is* derivable *if we can apply the rules of the structural operational semantics starting from this expression and arriving to $true$.*

Theorem 4.3 (Structural Operational Semantics = Operational View) *Let $M \in \mathrm{MS}(\sigma, \varepsilon)$ be a well-formed modular system. Then, for all τ-structures $\mathcal{B}_1$ and $\mathcal{B}_2$ (with $\tau \supseteq (\sigma \cup \varepsilon)$):*

$$\mathcal{B}_2 \in [\![M]\!](\mathcal{B}_1) \textit{ if and only if } (M, \mathcal{B}_1) \longrightarrow \mathcal{B}_2 \textit{ is derivable.}$$

Theorem 4.3 shows that, indeed, the operator we define for each modular system M (through our operational view on modular systems) precisely corresponds to the meaning of modular system M according to structural operational semantics. Note that both structural operational semantics

and our operational view on modular systems give a closer correspondence to the inner working of a modular system than the model-theoretical semantics. This is because, our model-theoretical semantics considers only the fixpoints of the operator associated to a modular system while our operational view (and our operational semantics) define the operator itself and can thus be used to talk about both the transient states of a modular system as well as its steady states (fixpoints). Further study of the transient states of modular systems is the subject of future research.

4.4 Expressive Power

The authors of [149] emphasized the importance of *capturing NP* and other complexity classes. The capturing property, say for NP, is of fundamental importance as it shows that, for a given language:

(a) *we can express all of NP* – which gives the user an assurance of universality of the language for the given complexity class,

(b) *no more than NP can be expressed* – thus solving can be achieved by means of constructing a universal polytime reduction (called *grounding*) to an NP-complete problem such as SAT or CSP.

In the context of modular systems, we also want to investigate the expressive power of the defined language. This section studies the complexity of modular systems in terms of a combination of both the model-theoretic view and the operational view towards modular systems.

In what follows, we first introduce several properties that a modular system may have as an operator. Examples of such properties are totality, determinacy, monotonicity and anti-monotonicity. We also prove several small results about these properties that will be useful in this chapter. Afterwards, we extend the complexity-theoretical concepts of polytime solvability, polytime checkability, etc. in a natural way to modular systems. Finally, we show our capturing result for modular systems.

Since we can view modular systems as operators, we can also classify them according to the properties (such as totality, determinacy, monotonicity, anti-monotonicity, etc.) of the operator associated with them.

Definition 4.14 (Total Modular Systems) *Let $M \in \mathrm{MS}(\sigma, \varepsilon)$ be a modular system and let τ be a set of vocabulary symbols such that $\tau \supseteq (\sigma \cup \varepsilon)$. Also, let C be a class of τ-structures. We say that* M is total *if the operator $[\![M]\!]$ is defined on all τ-structures $\mathcal{B}$ in C, i.e.,*

$$\mathcal{B} \in C \Rightarrow [\![M]\!]_\tau(\mathcal{B}) \neq \emptyset.$$

Our definition of totality is conceptually similar to [120]. We might omit mentioning the class of structures C either if it is obvious from the context or if the discussion holds for all classes of

structures.

Definition 4.15 (Deterministic Modular Systems) *For modular system M and sets of symbols τ and τ', we say M* is τ-τ'-deterministic *if for all structures $\mathcal{B}_1$ and $\mathcal{B}_2$, we have:*

$$\text{if } \mathcal{B}_1' \in [\![M]\!](\mathcal{B}_1), \mathcal{B}_2' \in [\![M]\!](\mathcal{B}_2) \text{ and } \mathcal{B}_1|_\tau = \mathcal{B}_2|_\tau \text{ then } \mathcal{B}_1'|_{\tau'} = \mathcal{B}_2'|_{\tau'}.$$

The following few basic propositions show how more properties about determinacy of a modular system can be obtained from other such properties.

Proposition 4.1 *For τ-τ'-deterministic modular system M:*

1. *If $\tau'' \supseteq \tau$, then M is also τ''-τ'-deterministic.*
2. *If $\tau'' \subseteq \tau'$, then M is also τ-τ''-deterministic.*

Proposition 4.2 *If M is both τ_1-τ_2-deterministic and τ_1'-τ_2'-deterministic, M is also $(\tau_1 \cup \tau_1')$-$(\tau_2 \cup \tau_2')$-deterministic.*

Definition 4.16 (Monotonicity and Anti-Monotonicity) *For modular system M and sets of symbols τ_1, τ_2 and τ_3, we say M* is τ_1-τ_2-τ_3-monotone *(resp.* τ_1-τ_2-τ_3-anti-monotone*) if for all structures $\mathcal{B}_1$ and $\mathcal{B}_2$, we have:*

$$\text{if } \mathcal{B}_1|_{\tau_1} \sqsubseteq_P \mathcal{B}_2|_{\tau_1} \text{ and } \mathcal{B}_1|_{\tau_2} = \mathcal{B}_2|_{\tau_2} \text{ then } \mathcal{B}_1'|_{\tau_3} \sqsubseteq_P \mathcal{B}_2'|_{\tau_3} (\text{resp. } \mathcal{B}_2'|_{\tau_3} \sqsubseteq_P \mathcal{B}_1'|_{\tau_3}).$$

where $\mathcal{B}_1' \in [\![M]\!](\mathcal{B}_1), \mathcal{B}_2' \in [\![M]\!](\mathcal{B}_1)$.

Proposition 4.3 *Let M be a τ_1-τ_2-τ_3-monotone or a τ_1-τ_2-τ_3-anti-monotone module. Then M is $(\tau_1 \cup \tau_2)$-τ_3-deterministic.*

Proof: We prove this for the monotone case. The other case is similar. Let $\mathcal{B}_1, \mathcal{B}_2 \in M$ be such that $\mathcal{B}_1|_{\tau_1 \cup \tau_2} = \mathcal{B}_2|_{\tau_1 \cup \tau_2}$. Then, (1) $\mathcal{B}_1|_{\tau_2} = \mathcal{B}_2|_{\tau_2}$, (2) $\mathcal{B}_1|_{\tau_1} \sqsubseteq_P \mathcal{B}_2|_{\tau_1}$, and, (3) $\mathcal{B}_2|_{\tau_1} \sqsubseteq_P \mathcal{B}_1|_{\tau_1}$. Also, let $\mathcal{B}_1' \in [\![M]\!](\mathcal{B}_1)$ and $\mathcal{B}_2' \in [\![M]\!](\mathcal{B}_2)$. Thus, by (1) and (2), we know $\mathcal{B}_1'|_{\tau_3} \sqsubseteq_P \mathcal{B}_2'|_{\tau_3}$ and, by (1) and (3), we have $\mathcal{B}_2'|_{\tau_3} \sqsubseteq_P \mathcal{B}_1'|_{\tau_3}$. Thus, $\mathcal{B}|_{\tau_3} = \mathcal{B}'|_{\tau_3}$. ∎

After defining total, deterministic, monotone and anti-monotone modular systems, we now define the complexity of a modular system as follows.

Recall that, in complexity theory, polynomial hierarchy is defined as follows: (i) $\Sigma_0^P = \Pi_0^P = \Delta_0^P := P$, and, (ii) $\Delta_{k+1}^P := \mathrm{P}^{\Sigma_k^P}$, $\Sigma_{k+1}^P := \mathrm{NP}^{\Sigma_k^P}$, and, $\Pi_{k+1}^P := \mathrm{coNP}^{\Sigma_k^P}$. Here, P, NP and coNP respectively refer to the set of polynomial time acceptable problems, non-deterministic polynomial time acceptable problems, and, non-deterministic polynomial time refutable problems.

Definition 4.17 (Σ_k^P-checkability, Δ_k^P-solvability) *Let $M \in \mathrm{MS}(\sigma, \varepsilon)$ be a modular system. We say that M* is Σ_k^P checkable *if the set M of structures is Δ_k^P decidable. Also, M* is Δ_k^P solvable *if there is a partial function F computable in Δ_k^P such that for all finite σ-structures $\mathcal{A}$: (1) $F(\mathcal{A})$ is defined if and only if there is $(\sigma \cup \varepsilon)$-structure $\mathcal{B} \in M$ expanding $\mathcal{A}$, and (2) if $F(\mathcal{A})$ is defined then $F(\mathcal{A}) \in M$ and $F(\mathcal{A})$ is the only structure in M that expands $\mathcal{A}$.*

Note that Δ_k^P solvability implies determinism. In what follows, we investigate the expressiveness of a modular system relative to the expressiveness of its primitive modules in two cases: (1) when both feedback and union operators are absent, and, (2) when at least one of feedback operator or union operator are present.

Theorem 4.4 that follows shows that when a modular system is formed by combining Δ_k^P solvable primitive modules using the projection and composition operators, i.e., feedback and union operators are not used, the modular system itself would also be Δ_k^P solvable. Theorem 4.5 shows that using the operators of union and feedback (either alone or together) to combine Δ_k^P solvable primitive modules enables us to express all Σ_k^P problems. Moreover, Theorem 4.5 also shows that our expressive power does not change even if we allow Δ_k^P checkable primitive modules. Thus, Theorem 4.5 characterizes the complexity and expressive power of union and feedback operators.

In the following, we view a problem as a class of structures rather than a set of binary strings (as is common in computational complexity). Viewing problems as classes of structures is very common in descriptive complexity and is also intuitively tied to the default definition through binary strings (as demonstrated by many standard results in descriptive complexity).

Theorem 4.4 (Expressiveness in the Absence of Feedback and Union Operators) *Let $\mathcal{K}$ be a problem over the class of finite structures closed under isomorphism. Then, the following are equivalent:*

1. *$\mathcal{K}$ is in Δ_k^P,*
2. *$\mathcal{K}$ is the models of a modular system where all primitive modules of M are Δ_k^P solvable and M does not use either the feedback operator or the union operator (i.e., M uses only primitive modules, composition operator and projection operator).*

Note that, in the following proof, we use operation $\mathcal{B}_1||\mathcal{B}_2$ to concatenate two structures $\mathcal{B}_1$ and $\mathcal{B}_2$ together. Be reminded that this operation is defined in Section 6.2.

Proof: (1) $\Rightarrow$ (2) is trivial. Just take M to be a primitive module accepting exactly $\mathcal{K}$.

(2) $\Rightarrow$ (1). Since M does not use the feedback operator, it is directional, i.e., there is a well-founded strict partial ordering $<$ on vocabulary symbols of M such that, the set of possible interpretations of

any given vocabulary symbol S depends only on the interpretation of vocabulary symbols S' with $S' < S$. This ordering is simply the ordering that is imposed by the input-output signature of all modules: $S_1 < S_2$ if and only if subsystem M' of M exists such that $S_1 \in \sigma_{M'}$ and $S_2 \in \varepsilon_{M'}$. Therefore, one can start with instance interpretations and compute (unique) interpretations of expansion predicates one by one (through application of primitive modules). One stops and rejects the input whenever any of the primitive modules does not accept the current computed interpretations. To formalize this argument, a simple induction on the structure of M suffices. The base case is when M is a primitive module. In this case, by assumption, M is Δ_k^P-solvable. There are also two inductive cases of $M := \pi_\tau(M')$ and $M := M_1 \rhd M_2$. In the former case, by induction hypothesis, M' is Δ_k^P solvable. Let F' be the partial function that witnesses Δ_k^P solvability of M'. Define $F(\mathcal{A}) := F'(\mathcal{A})|_\tau$. Function F can obviously be computed in Δ_k^P. Also, F is a partial function that witnesses Δ_k^P solvability of M. In the latter case, by induction hypothesis, M_1 and M_2 are Δ_k^P solvable. So, let F_1 and F_2 be the partial functions that witness this Δ_k^P solvability. Define $\mathcal{A}' := \mathcal{A}||(F_1(\mathcal{A})|_{\varepsilon_{M_1}})$ and $F(\mathcal{A}) := \mathcal{A}'||(F_2(\mathcal{A}'|_{\sigma_{M_2}})|_{\varepsilon_{M_2}})$. Obviously, F can be computed in Δ_k^P (because both F_1 and F_2 are computable in Δ_k^P). Moreover, F is not defined exactly when either F_1 is not defined or F_2 is not defined on the result of F_1. So, F is a partial function that witnesses the Δ_k^P solvability of M. ∎

Theorem 4.5 below shows that, in the presence of either the feedback operator or the union operator, our modular framework becomes much more expressive, i.e., it can express all of Σ_{k+1}^P despite its modules being all Δ_k^P-solvable. This is in contrast to Theorem 4.4 that discusses the case of loops and unions not being used. In fact, Theorem 4.5 shows much more by stating that, in the presence of these operators, regardless of whether you are limited to using only "easy" primitive modules or you are allowed to use complex primitive modules, one can always describe very complex problems. That is, we show that adding the feedback operator and our union operator cause a jump from one level of the polynomial hierarchy to the next, i.e., using only primitive modules of complexity Δ_k^P (level k of the polynomial hiearchy), our modular framework can express all search problems in Σ_{k+1}^P (the search problems in level $k+1$ of polynomial hierarchy).

Theorem 4.5 (Σ_{k+1}^P-Capturing over Finite Structures) *Let $\mathcal{K}$ be a problem over the class of finite structures closed under isomorphism. Then, the following are equivalent:*

1. *$\mathcal{K}$ is in Σ_k^P,*
2. *$\mathcal{K}$ is the models of a modular system M so that M only uses the feedback and projection operators and all primitive modules M' of M are $\sigma_{M'}$-$\varepsilon_{M'}$-deterministic, $\sigma_{M'}$-total, and Δ_k^P*

solvable,

3. *$\mathcal{K}$ is the models of a modular system M so that M does not use the feedback operator and all primitive modules M' of M are $\sigma_{M'}$-$\varepsilon_{M'}$-deterministic and Δ_k^P solvable.*
4. *$\mathcal{K}$ is the models of a modular system with Δ_k^P checkable primitive modules.*

As a consequence of Theorem 4.5 when $k = 0$, we know that using only primitive modules that are solvable in polynomial time, our modular framework is expressive enough to capture NP. In the proof below, we use HEX programs [64] that are similar to ASP programs extended with external atoms of form $\#g[I_1, \cdots, I_k](x_1, \cdots, x_n)$ that can be used in the body of the rules of a HEX program. Here, $I_1, \cdots, I_k$ are predicate symbols and $x_1, \cdots, x_n$ are variables that range over domain elements. Intuitively, an external atom $\#g[A_1, \cdots, A_k](a_1, \cdots, a_n)$ evaluates to true if external function g accepts the tuple $(a_1, \cdots, a_n)$ when $I_1, \cdots, I_k$ are respectively interpreted by $A_1, \cdots, A_k$. The reader is referred to [64] for more details on HEX programs.

Proof: (1) $\Rightarrow$ (2): To prove this direction, we give a modular system M that contains only one primitive module M'. Primitive module M' given in the proof satisfies all conditions of totality, determinacy and Δ_k^P solvability as required by the theorem statement. Module M feeds M''s output to part of its input and projects out some auxiliary vocabulary required by M'.

The proof in this direction follows the fact that, when allowing auxiliary vocabulary, ASP programs can express first order sentences (via Lloyd-Topor transformation). Thus, as FO MX captures NP over the class of finite structures, so do ASP programs (modulo the auxiliary vocabulary).

Now, consider a problem $\mathcal{K}$ in Σ_k^P with vocabulary σ, i.e., an isomorphism-closed set of finite σ-structures. Since $\mathcal{K} \in \Sigma_k^P$, $\mathcal{K}$ has a second order specification Φ so that Φ all the outermost second order quantifiers of Φ are existential and that Φ uses at most $k-1$ second-order quantifier alternations[3]. Now, let $\psi_1, \cdots, \psi_m$ be all the maximal subformulas of Φ that start with a second-order universal quantifier. Obviously, each ψ_i is decidable in Π_{k-1}^P and, so, also in Δ_k^P. Now, we obtain Φ' from Φ by replacing each subformula ψ_i of Φ with a higher-order atomic formula $A_i(I_1, \cdots, I_l, x_1, \cdots, x_n)$ where $I_1, \cdots, I_l$ are all the open (i.e., used but not quantified) relational symbols of ψ_i and $x_1, \cdots, x_n$ are all the open variables of ψ_i. Moreover, we define $A_i(I_1, \cdots, I_l, x_1, \cdots, x_n) := \psi_i(I_1, \cdots, I_l, x_1, \cdots, x_n)$. Obviously, Φ and Φ' are equivalent and all A_i's are decidable in Δ_k^P. Moreover, Φ', by construction, only uses existential second order

[3]It means that when formula Φ is represented as a labeled tree, every path from the root of this tree to one of its leaves has at most $k-1$ alternations between being a universally quantified node and being an existentially quantified node. Note that universal second-order quantifiers in negative positions are counted as existential quantifiers and vice versa.

quantifiers. Therefore, Φ' can be viewed as a model expansion task for first-order logic extended with atomic formulas $A_1, \cdots, A_m$.

By the above argument, there is also a HEX program P with instance vocabulary σ that accepts exactly those structures $\mathcal{B}$ with $\mathcal{B}|_\sigma \in \mathcal{K}$. In P, external function $\#A_i[I_1, \cdots, I_l](x_1, \cdots, x_n)$ is used instead of atom $A_i(I_1, \cdots, I_l, x_1, \cdots, x_n)$. As discussed, all such external functions can be computed in Δ_k^P. Moreover, since P is obtained through Lloyd-Topor transformation, we know that there is at most one atom in the head of all rules in P, i.e., P is a normal program and not a disjunctive one. Now, let $M' \in \mathrm{MS}(\sigma \cup \varepsilon_P, \varepsilon'_P)$ (with ε'_P being a set of new predicate symbols R' for each symbol $R \in \varepsilon_P$) be defined as follows. Given an instance structure $\mathcal{A}$, M' first computes the FLP reduct of P under $\mathcal{A}$, denoted as $P^{\mathcal{A}}$ and then, since P is normal, computes (in polynomial time and with access to external functions) and outputs the unique minimal model of $P^{\mathcal{A}}$. Since all operations of M' are polynomial time and external computation is in Δ_k^P, the whole procedure is in Δ_k^P complexity class.

Furthermore, M' is deterministic and total as required. Now, we define $M := \pi_\sigma(M'[\varepsilon_P = \varepsilon'_P])$. Observe that models of M are exactly those in $\mathcal{K}$.

(1) $\Rightarrow$ (3): Since $\mathcal{K} \in \Sigma_{k+1}^P$, there exists a set of new symbols ε and a verification procedure $V \in \Delta_k^P$ from $(\sigma \cup \varepsilon)$-structures to $\{0, 1\}$ such that $\mathcal{A} \in \mathcal{K}$ if and only if there exists a $(\sigma \cup \varepsilon)$-structure $\mathcal{B}$ that expands $\mathcal{A}$ and $V(\mathcal{B}) = 1$. Firstly, define σ' to be a set of new symbols R' for each symbol $R \in \sigma$ and define V' to be the same as V except that it takes $(\sigma' \cup \varepsilon)$-structures instead of $(\sigma \cup \varepsilon)$-structures. Now, define module M' such that $\sigma_{M'} := \sigma' \cup \varepsilon$ and $\varepsilon_{M'} := \emptyset$, i.e., M' has no output and thus its only role is to either accept or reject its input structure. M' accepts a $(\sigma' \cup \varepsilon)$-structure $\mathcal{B}$ in its input if (1) there exists $R \in \varepsilon$ such that $R^{\mathcal{B}} \neq \emptyset$, and, (2) either $V(\mathcal{B}) = 1$ or $V(\mathcal{B}') = 1$ where $\mathcal{B}'$ is the empty expansion of $\mathcal{A} := \mathcal{B}|_{\sigma'}$, i.e., $\mathcal{B}'$ is a $(\sigma' \cup \varepsilon)$-structure with $\mathcal{B}'|_{\sigma'} := \mathcal{A}$ and $R^{\mathcal{B}'} := \emptyset$ for all $R \in \varepsilon$.

Moreover, we define modules E and I such that $\sigma_E := \emptyset$, $\varepsilon_E := \varepsilon$, $\sigma_I := \sigma$, and, $\varepsilon_I := \sigma$. Module E always outputs a unique structure $\mathcal{A}$ such that $R^{\mathcal{A}} := \emptyset$ for all $R \in \varepsilon$ and module I just copies its input to its output, i.e., $(\sigma \cup \sigma')$-structure $\mathcal{B}$ belongs to I if and only if, for all $R \in \sigma$ and the corresponding $R' \in \sigma'$, we have $R^{\mathcal{A}} = R'^{\mathcal{A}}$. Finally, we define $M := \pi_\sigma((I \cup E) \rhd M')$.

Note that all primitive modules I, E and M' of M are deterministic and Δ_k^P solvable. Hence, we just need to show that, for all σ-structures $\mathcal{A}$, we have $\mathcal{A} \in M$ if and only if $\mathcal{A} \in \mathcal{K}$. This is not hard to prove because if $\mathcal{A} \in \mathcal{K}$ then there exists $(\sigma \cup \varepsilon)$-structure $\mathcal{B}$ with $V(\mathcal{B}) = 1$. Now, define $(\sigma \cup \sigma' \cup \varepsilon)$-structure $\mathcal{B}'$ such that (1) $\mathcal{B}'|_\sigma := \mathcal{B}|_\sigma$, (2) for all $R' \in \sigma'$ and the corresponding $R \in \sigma$, $R'^{\mathcal{B}'} := R^{\mathcal{B}}$, and, (3) $\mathcal{B}'|_\varepsilon := \mathcal{B}|_\varepsilon$ if and only if there exists $R \in \varepsilon$ with $R^{\mathcal{B}} \neq \emptyset$. Note that

$\mathcal{B}'|_{\sigma\cup\sigma'} \in I$ and so $\mathcal{B}' \in (I \cup E)$. Also, since $\mathcal{B}'|_{\sigma'\cup\varepsilon} \in M'$, $\mathcal{B}' \in ((I \cup E) \rhd M')$. Therefore, $\mathcal{A} \in M$. On the other hand, if $\mathcal{A} \in M$ then there exists a $(\sigma \cup \sigma' \cup \varepsilon)$-structure $\mathcal{B}' \in ((I \cup E) \rhd M')$. Thus, $\mathcal{B}'|_{\sigma'\cup\varepsilon} \in M'$ and $\mathcal{B}'|_{\sigma\cup\sigma'} \in (I \cup E)$. Using the first part, we know $R^{\mathcal{B}'} \neq \emptyset$ for some $R \in \varepsilon$. Hence, $\mathcal{B}'|_{\varepsilon} \notin E$ and, so, $\mathcal{B}'|_{\sigma\cup\sigma'} \in I$, i.e., interpretation of σ' is copied from the interpretation of σ. Thus, by $\mathcal{B}|_{\sigma\cup\varepsilon} \in M'$, we know that there exists an expansion of $\mathcal{A}$ that is accepted by verification procedure V. Thus, $\mathcal{A} \in \mathcal{K}$.

(2) $\Rightarrow$ (4), and, (3) $\Rightarrow$ (4): This direction is trivial because, in both cases (2) and (3), M uses only Δ_k^P solvable primitive modules and, thus, only Δ_k^P checkable primitive modules.

(4) $\Rightarrow$ (1): Let M be a modular system whose models coincide with $\mathcal{K}$ and whose primitive modules are Δ_k^P checkable. Then, $\mathcal{K}$ is in Σ_{k+1}^P because one can nondeterministically guess all the interpretations of expansion symbols of M (the set of these symbols is equal to the disjoint union of the expansion vocabularies of all M's primitive modules) and then use Δ_k^P checkability of M's primitive modules to check if this is a good guess (according to the modules, and thus according to the system itself). ∎

Theorem 4.5 demonstrates the additional power that the feedback operator has brought to us. Its proof assumes that modules are described in languages with the ability to manipulate input programs and sets of atoms, and to compute fixpoints. Examples of such languages are those that capture P in the presence of ordering relation over domain elements, or the like. However, note that, in our model-theoretic view, the language that modules are described in is not important at all.

Note that Theorem 4.5 (with $k = 0$) shows that when basic modules are restricted to polytime checkable modules, the modular system's expressive power is limited to NP. Of course, if we do not restrict the computational power of primitive module, the modular framework can represent problems that are not even Turing computable. As an example, one can encode Turing machines as finite structures and have modules that accept a finite structure if and only if it corresponds to a halting Turing machine.

Theorem 4.5 shows that the feedback operator causes a jump in expressive power from Δ_k^P to Σ_{k+1}^P. The proof uses a translation from HEX programs to modular systems. The following running example elaborates more in this direction for the case of $k = 0$ (in this specific case, HEX programs can be simply replaced by ASP programs which are more limited).

Example 4.7 (Stable Model Semantics) *Let P be a normal logic program. We know S is a stable model for P iff $S = Dcl(P^S)$ where P^S is the reduct of P under set S of atoms (a positive program)*

and Dcl computes the deductive closure of a positive program, i.e., the smallest set of atoms satisfying it. Now, let $M_1(S, P, Q)$ be the module that given a set of atoms S and ASP program P computes the reduct Q of P under S. Observe that M_1 is $\{S\}$-total and $\{S\}$-$\{P\}$-$\{Q\}$-anti-monotone, and polytime solvable. Also, let $M_2(Q, S')$ be a module that, given a positive logic program Q, returns the smallest set of atoms S' satisfying Q. Again, M_2 is $\{Q\}$-total, $\{Q\}$-$\{\}$-$\{S'\}$-monotone and polytime solvable. However, $M := \pi_{\{P,S\}}((M_1 \rhd M_2)[S = S'])$ is a module which, given ground ASP program P, returns all and only the stable models of P. Therefore, the NP-complete problem of finding a stable model for a normal logic program is defined by combining total, deterministic, polytime solvable, and monotone or anti-monotone modules.

Example 4.7 shows that stable models can be naturally modeled in the context of modular systems. As we will see later in this chapter, this phenomenon is not accidental but a consequence of anti-monotone loops (feedbacks). Moreover, we already know that the modular framework does not impose minimality constraint on the solution to its modules (while stable model semantics does). Thus, in the presence of only polytime solvable modules, our framework can define sets of structures that cannot be defined in ASP.

Moreover, the modular framework introduces a new way of combining ASP programs. Each ASP program can be represented as a module and these modules can be combined through the algebraic expressions we introduced. This is a new way of combining ASP programs since neither circular (through feedback) nor disjunctive (through union) combinations of ASP programs have not been used before. Theorem 4.5 characterizes the expressive power of the resulting formalism.

4.5 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). We start by a very simple but inefficient algorithm in Section 4.5.1 and then revise this algorithm and propose a more efficient algorithm in Section 4.5.5.

4.5.1 Naive Modular Model Expansion Algorithm

The first algorithm that we propose in this section is a naive algorithm that, given an instance σ-structure $\mathcal{A}$ with finite domain, guesses a $(\sigma \cup \varepsilon)$-structure $\mathcal{B}$ non-deterministically so that $\mathcal{B}$ is an expansion of $\mathcal{A}$ and $\mathcal{B} \in M$. The simple and naive algorithm that we propose here has two

purposes. First, it gives a concrete algorithm that partly witnesses what we proved in Theorem 4.5, i.e., a modular system with all its primitive modules decidable in Δ_k^P, can be solved in Σ_{k+1}^P. Second, the naive algorithm proposed in this section paves the way for the more complex and more efficient algorithm that will be given in Section 4.5.5.

Algorithm 1 is specialized for a modular system M that has the primitive modules $M_1, \cdots, M_k$. Each primitive module M_i is assumed to have an instance vocabulary σ_i and an expansion vocabulary ε_i. Moreover, each primitive module is associated with a decision procedure D_i.

Data: Finite σ-structure $\mathcal{A}$
Result: Structure $\mathcal{B}$ that expands $\mathcal{A}$ and is in M
begin
  Let $\mathcal{B}$ be a $(\sigma \cup \varepsilon)$-structure s.t. $R^{\mathcal{B}} = R^{\mathcal{A}}$ for all $R \in \sigma$ and $R^{\mathcal{B}} = \emptyset$ otherwise;
  foreach *n-ary predicate symbol* $R \in \varepsilon$ **do**
    foreach *n-tuple* $\langle \tau_1, \cdots, \tau_n \rangle \in [dom(\mathcal{A})]^n$ **do**
      Non-determinstically choose $b \in \{0, 1\}$;
      if $b = 1$ **then** Add $\langle \tau_1, \cdots, \tau_n \rangle$ to $R^{\mathcal{B}}$;
      else Do nothing ;
  if *all decision procedures* D_i *accept* $\mathcal{B}|_{(\sigma_i \cup \varepsilon_i)}$ **then** Accept and return $\mathcal{B}$;
  else Reject ;
end

Algorithm 1: Guess and Check Algorithm for Modular Model Expansion

Algorithm 1 takes a finite instance structure $\mathcal{A}$ (i.e., the domain of $\mathcal{A}$ is finite) and uses nondeterminism to guess all different expansions $\mathcal{B}$ of $\mathcal{A}$. Then, each such $\mathcal{B}$ is checked against all primitive modules and if all of them are satisfied with the current guess, $\mathcal{B}$ is accepted and otherwise rejected. However, as one can never reasonably expect a solve a simple guess-and-check procedure to become an efficient propositional satisfiability solver, one can not expect Algorithm 1 to be practical either. Therefore, in the next sections, we develop a more efficient Algorithm that uses the decision procedures of primitive modules for more than just checking the final answer. As we will see in the next sections, each decision procedure (that will be called an oracle later on) is expected to provide more than a yes/no answer. This way the oracles (decision procedures) "assist" an external solver in finding a model (if one exists). There, we start from an empty expansion of $\mathcal{A}$ (i.e., a partial structure which contains no information about the expansion predicates), and then we gradually extend the current partial structure with new information (through an interaction with the oracles of the given modular system) until we either find a model that satisfies the modular system or we conclude that none exists. To model this procedure, we first define the notion of a partial

structure.

4.5.2 Partial Structures

Recall that a structure is a domain together with an interpretation of the associated vocabulary that here consists of a finite set of predicate and function symbols (including null-ary functions or constants). An interpretation of an n-ary relational symbol R in a structure $\mathcal{A}$ is a subset of $[dom(\mathcal{A})]^n$. However, sometime, we want to express the lack of knowledge about membership of a particular $\bar{a}$ in $R^{\mathcal{A}}$, i.e., instead of definitely answering whether $\bar{a} \in R^{\mathcal{A}}$ or not, we sometimes want to express our lack of knowledge about this relationship. In order to do so, we use *partial structures*. This way, we can gradually accumulate knowledge about structures. This is in contrast with how Algorithm 1 worked, i.e., to generate the structure as a whole and check it once everything is known about the structure.

Definition 4.18 (Partial Structure) *$\mathcal{B}$ 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 $R \in \tau_p$, $\mathcal{B}$ interprets R using two sets R^+ and R^- such that $R^+ \cap R^- = \emptyset$, and $R^+ \cup R^- \neq (dom(\mathcal{B}))^n$.*

In such cases, we call τ_p the partial vocabulary *of $\mathcal{B}$. Also, we say that $\mathcal{B}$ is* total *if $\tau_p = \emptyset$.*

Definition 4.19 (Extension for Partial Structures) *For two partial structures $\mathcal{B}$ and $\mathcal{B}'$ over the same vocabulary and domain, we say that $\mathcal{B}'$* extends *$\mathcal{B}$ if $\mathcal{B}'$ has at least as much information as $\mathcal{B}$ does (and possibly more), i.e., for τ'_p-partial structure $\mathcal{B}'$ and τ_p-partial structure $\mathcal{B}$ over the same vocabulary and domain, $\mathcal{B}'$ extends $\mathcal{B}$ if:*

$$
\begin{aligned}
&\tau'_p \subseteq \tau_p, \\
&\text{for all } R \in \tau \backslash \tau_p, \text{ we have: } R^{\mathcal{B}'} = R^{\mathcal{B}}, \\
&\text{for all } R \in \tau_p \backslash \tau'_p, \text{ we have: } R^{+^{\mathcal{B}}} \subseteq R^{\mathcal{B}'} \text{ and } R^{-^{\mathcal{B}}} \cap R^{\mathcal{B}'} = \emptyset, \text{ and,} \\
&\text{for all } R \in \tau_p \cap \tau'_p, \text{ we have: } R^{+^{\mathcal{B}}} \subseteq R^{+^{\mathcal{B}'}} \text{ and } R^{-^{\mathcal{B}}} \subseteq R^{-^{\mathcal{B}'}}.
\end{aligned}
$$

Example 4.8 *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 $R^{-^{\mathcal{B}}} = \{\langle 1\rangle\}$.*

If a partial structure $\mathcal{B}$ has enough information to satisfy or falsify a formula ϕ, then we say $\mathcal{B} \models \phi$, or $\mathcal{B} \models \neg\phi$, respectively. Note that for partial structures, $\mathcal{B} \models \neg\phi$ and $\mathcal{B} \not\models \phi$ may be different, i.e., there exist cases where $\mathcal{B} \not\models \phi$ and $\mathcal{B} \not\models \neg\phi$. 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 $R \in \varepsilon$.

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 4.20 (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 $\mathcal{B}$.*

4.5.3 Requirements on the Modules

As expressed in the introduction, there is practical need to solve complex computational tasks in a modular way so that full access to a complete axiomatization of some modules 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" track 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 4.21 (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 4.22 (Solver Language) *For a language $\mathcal{L}$ with structural models, 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 properties that 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 $\Gamma \models_L A \supset B$. Here, the resolution theorem guarantees that, once an advice of form $Pre \supset Post$ is added to the solver, and 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 4.23 (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. Here, as also described in Section 4.4, we adopt the definition of a problem as a set of structures, similar to what is done in finite model theory and descriptive complexity.

Definition 4.24 (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 4.9 (Graph 3-coloring: Certificates) *Consider Example 2.1 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 4.25 (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 4.4 to prove the correctness of our algorithm.

Example 4.10 (Graph 3-coloring: Reasons and Advice) *Consider the graph 3-coloring example of Example 2.1. 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 partial structure $\mathcal{B} = (V^{\mathcal{G}}; E^{\mathcal{G}}, R^{\mathcal{B}}, B^{\mathcal{B}}, G^{\mathcal{B}})$ 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 axioms in a well-studied logic with an efficient solver for that logic, we can use these *axioms together with the associated solver* in order to obtain a low-cost and relatively efficient oracle for the module. For example, we often have existing efficient Valid Acceptance Procedures used in solver constructions of well-studied languages, e.g., Well-Founded Model computation for the language of ASP, Arc-Consistency checking for the language of CP, Theory Propagation for various languages of SMT theories, a lifted version of Unit Propagation [200]

for the language of 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 obtain a module-specific oracle by running a CP solver on a partial model for that module plus the CP axiomatization of that module. The module-specific oracle defined this way can use CP propagation techniques implemented in the CP solver to obtain some advice that it can return.

Moreover, using these well-defined advice generation/verification techniques for such well-studied languages (such as FO, ASP, CP, SMT, ILP, etc.), we can construct reasonably efficient *generic oracles* for those languages, i.e., oracles that can work on all specifications in a language rather than specific axiomatizations. However, one should always note that generic oracles are not the strongest oracles possible. Application-specific oracles that use the intuition of a module designer to derive intelligent advices about a partial model (or to intelligently verify a partial structure) of the module are often stronger than their generic counterparts (i.e., the generic advice/verification procedure). Similarly, the intuition of a module designer is also very important when deciding between different possible reasons of rejection. Therefore, although generic oracles offer a low-cost and reasonably efficient approach to obtain oracles, we believe that application-specific oracles should be preferred.

4.5.4 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 4.26 (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 4.4 gives the exact correspondence in this regard.

4.5.5 Lazy Modular Model Expansion Algorithm

In this section, we present a deterministic and iterative algorithm schema to solve model expansion tasks for modular systems. This algorithm schema can be instantiated to work for a particular modular system M by specifying all the necessary CCAV oracles for primitive modules and also instantiating S with a complete online solver. Every such specialization of Algorithm 2 takes a σ-structure $\mathcal{A}$ and returns an expansion $\mathcal{B}$ of $\mathcal{A}$ such that $\mathcal{B} \in M$ (if such $\mathcal{B}$ exists). Algorithm 2 works by accumulating reasons and advice from oracles and gradually converging to a solution to the problem.

Algorithm 2 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. Note that this simple approach respects all the operators defined for the modular system except the union operator.

Proposition 4.4 (Correctness) *Algorithm 2 is sound and complete [4] for finite structures, i.e., given a modular system M with CCAV oracles, a complete online solver S and a finite instance structure $\mathcal{A}$:*

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

```
Data: Input structure 𝒜
Result: Structure ℬ that expands 𝒜 and is in M
begin
    /* We assume that each primitive module M_i is associated with a CCAV oracle.   */
    /* We further assume that the solver S is a complete online solver               */
    Initialize the solver S using the empty expansion of 𝒜 ;
    while TRUE do
        Let R be the state of S ;
        if R = ⟨UNSAT⟩ then return Unsatisfiable ;
        else if R = ⟨SAT, ℬ⟩ then
            Add all the advice from oracles wrt ℬ to S ;
            if M does not accept ℬ then
                Find a module M_i in M such that M_i does not accept ℬ|_vocab(M_i) ;
                Add the reason given by oracle O_i to S ;
            else if ℬ is total then return ℬ ;
end
```

Algorithm 2: Lazy Modular Model Expansion Algorithm Schema

1. *If Algorithm 2 returns $\mathcal{B}$, then $\mathcal{B} \in M$,*

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

3. *Algorithm 2 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 4.5 (Time Complexity) *For all modular systems M, there is $k \in \mathbb{N}$ such that for each finite structure $\mathcal{A}$, Algorithm 2 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 4.4 plus the fact that $3^{O(|dom(\mathcal{A})|^k)}$ different partial interpretations exist. ∎

4.6 Case Studies: Existing Frameworks

This section is part of a joint work with Xiongnan (Newman) Wu that is also reported in his Master's Thesis [214]. 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 4.1 *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 $\neg l \in \neg S$.

4.6.1 Modelling DPLL(T)

The DPLL(T) [158] 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 [158].

In this section, the the following MX task is used as a running example to show how Algorithm 2 models the DPLL(T) system.

Example 4.11 (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 4.11 using the DPLL(T) system with the theory T being the Theory of Difference Logic [158].

Example 4.12 (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:

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

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

In the real world, the DPLL(T) system works on the propositional level and the axiomatization 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:

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

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

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:

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

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 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

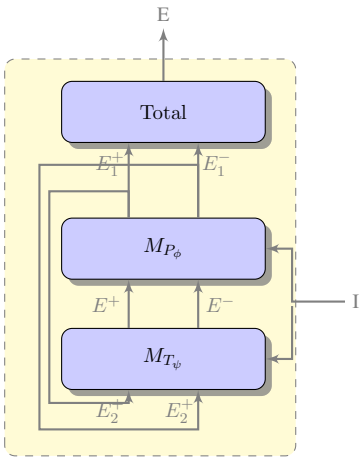

Figure 4.5: Modular System $DPLL(T)_{\phi\wedge\psi}$ Representing the DPLL(T) System on Input Formula $\phi \wedge \psi$.

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 2 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 4.5, 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 4.13 (Disjoint Scheduling Continued) *Continuing our running example, the assignment (4.5) is equivalent to the following partial structure $\mathcal{B}$:*

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

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 4.14 (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) [158].

Example 4.15 (Disjoint Scheduling Continued) *Continuing our running example, consider the case where $E_2^{+\mathcal{B}} = \{after(1), after(2), before(1), before(2)\}$ 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 formulas, 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 4.16 (Disjoint Scheduling Continued (O_P, O_T and O_{TOTAL})) *Let ϕ and ψ in Figure 4.5 be, respectively, the propositional part and theory part of the specification in Example 4.12. Let the structure $\mathcal{B}$ contain the same set of partial assignments as the one in Example 4.12, 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.[5] Clearly, O_{TOTAL} is a CCAV oracle.

Oracle O_T accepts a partial structure $\mathcal{B}$ iff it does not falsify the constraints described 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 $\{E_2^+(d) \supset E^+(d) \mid d \in D, \mathcal{B} \models E_2^+(d), \mathcal{B} \not\models E^+(d)\} \cup \{E_2^-(d) \supset E^-(d) \mid d \in D, \mathcal{B} \models$

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

$E_2^-(d), \mathcal{B} \not\models E^-(d)\}$ as the set of advices.[6] 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 $D_1 \subseteq D$, $D_2 \subseteq D$, $\emptyset \subsetneq D_3 \subseteq D$, $\phi \models \bigvee_{d\in D_1} \neg d \vee \bigvee_{d\in D_2} d$ 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 4.6 *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 2 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 [170]. The DPLL(Agg) module [33] 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.

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

4.6.2 Modelling ILP Solvers

Integer Linear Programming solvers solve optimization problems. In this chapter, 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 2 models branch-and-cut based ILP solvers.

Example 4.17 (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 4.17 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 [162] 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 $\{C^i \cup \{x_i \leq \lfloor v_i \rfloor\}, C^i \cup \{x_i \geq \lceil v_i \rceil\}\}$.

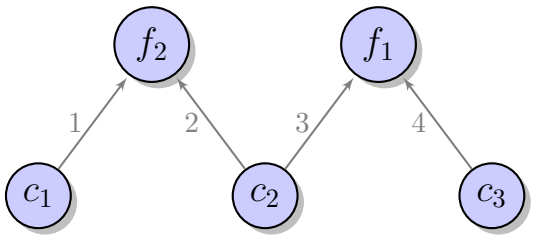

Figure 4.6: Facility Opening Problem Instance.

In order for the branch-and-cut algorithm above to solve the problem in Example 4.17, we must describe the example using a set of linear inequalities as follows.

Example 4.18 (Facility Opening (Specification, Instance, Ground Program))
We use the following high level ILP specification to represent the facility opening problem in Example 4.17.

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

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 4.18. In Figure 4.18, 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:

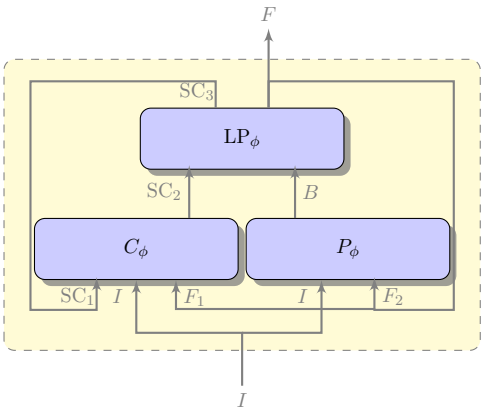

Figure 4.7: Modular System Representing an ILP Solver.

$$
\begin{aligned}
&U_{1,1} \leq 0, && U_{1,2} \leq 1, && \\
&U_{2,1} \leq 1, && U_{2,2} \leq 1, && \\
&U_{3,1} \leq 1, && U_{3,2} \leq 0, && \\
&U_{1,1} + U_{1,2} \geq 1, && U_{2,1} + U_{2,2} \geq 1, && U_{3,1} + U_{3,2} \geq 1, \\
&U_{1,1} - O_1 \leq 0, && U_{1,2} - O_2 \leq 0, && \\
&U_{2,1} - O_1 \leq 0, && U_{2,2} - O_2 \leq 0, && \\
&U_{3,1} - O_1 \leq 0, && U_{3,2} - O_2 \leq 0, && \\
&0.5\, O_1 + 1.5\, O_2 + U_{1,2} + 3\, U_{2,1} + 2\, U_{2,2} + 4\, U_{3,1} \leq 10, && && \\
&0 \leq U_{1,1}, U_{1,2}, U_{2,1}, U_{2,2}, U_{3,1}, U_{3,2}, O_1, O_2 \leq 1. && &&
\end{aligned}
$$

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.

$$
\begin{aligned}
&\textit{Partitioning: } U_{2,1} \leq 0 \vee U_{2,1} \geq 1, \\
&\textit{Cutting plane: } U_{2,1} + U_{2,2} \leq 1.
\end{aligned}
$$

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 4.7 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 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 4.19 (Facility Opening Problem Continued (O_p and O_c))
Let ϕ in Figure 4.7 be the specification shown in Example 4.18. Let $F^{\mathcal{B}}$ contain the same non-integral solution as the one in Example 4.18, 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 4.18, 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 $B(\text{"}U(2,1) \leq 0\text{"}) \vee B(\text{"}U(2,1) \geq 1\text{"})$ [7] ; and the cutting plane method can be modelled by O_c rejecting $\mathcal{B}$ with the reason $SC_2(\text{"}U(2,1) + U(2,2) \leq 1\text{"})$.

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

Example 4.20 (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 4.19, but let $B^{\mathcal{B}} = \{\text{"}U(2,1) \leq 0\text{"}\}$ and $SC_2 = \{\text{"}U(2,1) + U(2,2) \leq 1\text{"}\}$. 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 $B(\text{"}U(2,1) \leq 0\text{"}) \supset F(\text{"}U(2,1)\text{"}) \leq 0$ (for violating the partitioning constraint), or being $SC_2(\text{"}U(2,1) + U(2,2) \leq 1\text{"}) \wedge F_1(\text{"}U(2,2)\text{"}) > 1 \supset F_1(\text{"}U(2,1)\text{"}) < 0$ (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 $B(\text{"}F_2(x) \leq \lfloor v \rfloor\text{"}) \vee B(\text{"}F_2(x) \geq \lceil v \rceil\text{"})$, 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 $\bigwedge_i F_1(x_i) = v_i \supset \bigwedge_{c \in SC_1 \Delta SC_2}(SC_1(c) \iff SC_2(c))$ 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.

---

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

Proposition 4.7 *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 2 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.

4.6.3 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 [20], 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 [143], 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 [84] 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 [84], a partial grounding is applied before running the algorithm, thus, the algorithm in [84] 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 [82]. 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 [85] and constraint propagation in CP [168]. If a conflict occurs (during either unit propagation or constraint propagation), a nogood containing the corresponding unique implication point (UIP) [147, 142] is learned [8] 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 2 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 4.21 (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 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 4.21 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.

[8] Practical CP solvers do not provide reasons for rejecting partial structures. This issue is dealt with in [84] by wrapping CP solvers with a conflict analysis mechanism to compute nogoods based on the first UIP scheme.

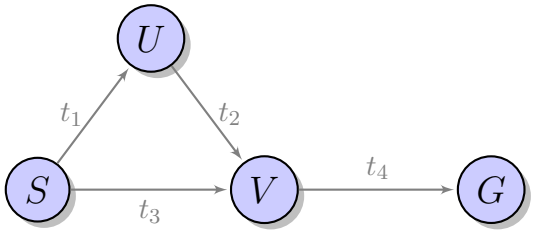

Figure 4.8: Facility Opening Problem Instance.

Figure 4.9: Planning with Cumulative Scheduling Problem Instance.

Example 4.22 (Planning Continued (Specification, Instance, Ground Program))
Following CASP specification is used to represent the planning with cumulative scheduling problem [9].

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

Consider the instance shown in figure 4.22 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.,* $CS^{\mathcal{A}} = \{(t_1, S, U), (t_2, U, V), (t_3, S, V), (t_4, V, G)\}$. *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:*

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

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

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

The CASP system solves the ground program 4.9 in a similar way to the DPLL(T) system described in Section 4.6.1: 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:

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

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 2 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 4.5 (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 4.5, 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 [85] on ϕ under $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}c} \cup E^{+\mathcal{B}} \cup \neg E^{-\mathcal{B}}$.

Example 4.23 (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 [168] for complete details on different propagation techniques.

Example 4.24 (Planning with Cumulative Scheduling Continued)
Continuing our running example, now assume that $E_2^{+\mathcal{B}}$ contains $do(t_4)$ as discussed in Example 4.23. 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 4.6.1.

As a CDNL-like technique is also used in SMT solvers, the above algorithm is modelled similarly to Section 4.6.1. 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 4.6.1.

Example 4.25 (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 4.22, 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 4.8 *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 2 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 $\phi \wedge \psi$.

4.7 Approximating Solutions to Modular Systems

So far, we introduced modular systems and talked about their expressive power. Also, we gave two algorithms to solve modular systems: the naive guess and check algorithm and the lazy modular solving algorithm. We also described how our lazy solving algorithm is related and, indeed, motivated by works in the solvers that are a specific to a language or a combination of a few languages.

In this Section, we want to go beyond what we had before and improve our lazy solving algorithm with the power to approximate the solutions to a modular system without excessive use of the

underlying solver. This way, we will be able to hugely reduce the search space of the underlying solver and find a solution to a modular system faster. To do so, we start with some simple properties about extending monotonicity and anti-monotonicity to complex modules. We prove that, in the presence of loops and monotone or anti-monotone primitive modules, the combined systems satisfy many interesting properties such as existence of smallest solutions or minimality of solutions. We then develop methods for intelligently reducing the candidate solution space.

4.7.1 Approximation Procedures for Modular Systems

Almost all practical solvers use some kind of propagation technique. However, in a modular system, propagation is not possible in general because nothing is known in advance about a module. Here, we define how and under what conditions can we approximate the solutions to a modular system. As it turns out, it suffices to know only very general information about modules such as their totality and monotonicity or anti-monotonicity. We first start by a few propositions on how totality, monotonicity and anti-monotonicity extend from simpler modular systems to more complex modular systems. Then, we state Theorems 4.6, 4.13 and 4.9 that constitute the main body of our approximation procedures for modular systems. As we explain later on, our proposed approximation procedures closely correspond to the modular equivalent of computing least/greatest fixpoints and well-founded models in logic programs.

Proposition 4.9 *Let M be a τ_1-τ_2-τ_3-monotone (resp. anti-monotone) module. Then:*

1. *If $\tau' \subseteq \tau_1$ then M is also a τ'-$(\tau_2 \cup (\tau_1 \backslash \tau'))$-$\tau_3$-monotone (resp. anti-monotone) module.*
2. *For a set ν of symbols such that $\tau_3 \cap \nu = \emptyset$, we have M is also $(\tau_1 \cup \nu)$-τ_2-τ_3-monotone (resp. anti-monotone).*
3. *For a set ν of symbols, we have that M is also τ_1-$(\tau_2 \cup \nu)$-τ_3-monotone (resp. anti-monotone).*
4. *If $\tau' \subseteq \tau_3$ then M is also a τ_1-τ_2-τ'-monotone (resp. anti-monotone) module.*

Proposition 4.10 *Let M be a module that is both τ_1-τ_2-τ_3-monotone and τ_1'-τ_2'-τ_3'-monotone (resp. τ_1-τ_2-τ_3-anti-monotone and τ_1'-τ_2'-τ_3'-anti-monotone) such that $(\tau_1 \cup \tau_1') \cap (\tau_3 \cup \tau_3') = \emptyset$. Then, M is also $(\tau_1 \cup \tau_1')$-$(\tau_2 \cup \tau_2')$-$(\tau_3 \cup \tau_3')$-monotone (resp. $(\tau_1 \cap \tau_1')$-$(\tau_2 \cup \tau_2')$-$(\tau_3 \cup \tau_3')$-anti-monotone).*

Proposition 4.11 ((Anti-)Monotonicity Preservation) *For τ_1-τ_2-τ_3-monotone (resp. anti-monotone) modular system M and general modular system M', we have:*

1. *$M \rhd M'$ is τ_1-τ_2-τ_3-monotone (resp. anti-monotone).*
2. *$M' \rhd M$ is τ_1-τ_2-τ_3-monotone (resp. anti-monotone).*

3. *If M' is ν-τ_2-deterministic for some ν, then $M' \rhd M$ is τ_1-ν-τ_3-monotone (resp. anti-monotone).*
4. *If $\tau_1 \cup \tau_2 \subseteq \nu$ then $\Pi_\nu M$ is τ_1-τ_2-$(\nu \cap \tau_3)$-monotone (resp. anti-monotone).*
5. *$M[S_1 = S_2]$ is τ_1-τ_2-τ_3-monotone (resp. anti-monotone)*

Proposition 4.12 (Monotonicity under Composition) *For modular systems M and M' and vocabularies τ_1, τ_1', τ_2, τ_2', τ_3 and τ_3' such that $\tau_1' \subseteq \tau_3$:*

1. *If M is τ_1-τ_2-τ_3-monotone and M' is τ_1'-τ_2'-τ_3'-monotone, $M \rhd M'$ is τ_1-$(\tau_2 \cup \tau_2')$-τ_3'-monotone.*
2. *If M is τ_1-τ_2-τ_3-anti-monotone and M' is τ_1'-τ_2'-τ_3'-monotone, $M \rhd M'$ is τ_1-$(\tau_2 \cup \tau_2')$-τ_3'-anti-monotone.*
3. *If M is τ_1-τ_2-τ_3-monotone and M' is τ_1'-τ_2'-τ_3'-anti-monotone, $M \rhd M'$ is τ_1-$(\tau_2 \cup \tau_2')$-τ_3'-anti-monotone.*
4. *If M is τ_1-τ_2-τ_3-anti-monotone and M' is τ_1'-τ_2'-τ_3'-anti-monotone, $M \rhd M'$ is τ_1-$(\tau_2 \cup \tau_2')$-τ_3'-monotone.*

Proof: We prove the first case. The rest is similar. For $P := M \rhd M'$, consider structures $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_1' \in [\![M_1]\!](\mathcal{B}_1)$ and $\mathcal{B}_2' \in [\![M_2]\!](\mathcal{B}_2)$ such that $\mathcal{B}_1|_{\tau_2 \cup \tau_2'} = \mathcal{B}'|_{\tau_2 \cup \tau_2'}$ and $\mathcal{B}|_{\tau_1} \sqsubseteq \mathcal{B}'|_{\tau_1}$. Since M is monotone, we have $\mathcal{B}|_{\tau_3} \sqsubseteq \mathcal{B}'|_{\tau_3}$. So, as $\tau_1' \subseteq \tau_3$, we also have $\mathcal{B}|_{\tau_1'} \sqsubseteq \mathcal{B}'|_{\tau_1'}$. Also, as M' is monotone, we have $\mathcal{B}|_{\tau_3'} \sqsubseteq \mathcal{B}'|_{\tau_3'}$. ∎

These properties give us ways of deriving that a complex modular system is monotone or anti-monotone by looking at similar properties of basic constraint modules. For instance, for our previous example on stable model semantics, we have:

Example 4.26 (Composition in ASP Programs) *Modules M_1 and M_2 in Example 4.7 are respectively $\{S\}$-$\{P\}$-$\{Q\}$-anti-monotone and $\{Q\}$-$\{\}$-$\{S'\}$-monotone. So, by Proposition 4.12, $M' := M_1 \rhd M_2$ is $\{S\}$-$\{P\}$-$\{S'\}$-anti-monotone.*

The rest of this section considers the important case of monotone or anti-monotone loops, i.e., monotone or anti-monotone modules under the feedback operator. Note that, although our theorems concern modules feeding their outputs back to their inputs, these modules are usually not primitive modules, but composite modules whose monotonicity or anti-monotonicity is derived by our previous propositions.

Theorem 4.6 (Smallest/Greatest Solution) *Let M be a $(\tau \cup \{S\})$-total and $\{S\}$-τ-$\{R\}$-monotone modular system and $M' := M[S = R]$. Then, for a fixed interpretation to τ, M' has exactly one smallest solution and exactly one greatest solution with respect to predicate symbol R.*

Proof: Standard Tarski proof. ∎

Theorem 4.6 relates smallest/greatest solutions of monotone loops in modular systems to least-/greatest fixpoints of monotone operators. Therefore, many natural problems such as transitivity or connectivity are smallest solutions of some monotone modules under feedbacks. However, Theorem 4.6 only states that such smallest/greatest solutions exist and is unique and it should be noted that modular systems can have other models that are neither the smallest nor the greatest solution. The special cases of smallest and greatest solutions are used to prune the candidate solution space, i.e., candidate solutions that either do not extend the smallest solution or that are not extended by the greatest solution can be safely discarded. Theorem 4.6 can also be extended to a more general case as follows.

Theorem 4.7 (Optimal Bounding to Solutions of Positive Feedbacks) *Let M be a $(\tau \cup \{S\})$-total and $\{S\}$-τ-$\{R\}$-monotone modular system and $M' := M[S = R]$. Also, let $\mathcal{B}_L$ and $\mathcal{B}_U$ be two structures such that $\mathcal{B}_L|_{vocab(\mathcal{B}_L)\setminus\{R,S\}} \sqsubseteq \mathcal{B}_U|_{vocab(\mathcal{B}_U)\setminus\{R,S\}}$ and $S^{\mathcal{B}_L} = R^{\mathcal{B}_L} \sqsubseteq R^{\mathcal{B}_U} = S^{\mathcal{B}_U}$. Then, if M' has any solution $\mathcal{B}$ with $\mathcal{B}|_\tau = \mathcal{B}_L|_\tau$ and $R^{\mathcal{B}_L} \subseteq R^{\mathcal{B}} \subseteq R^{\mathcal{B}_U}$, then the structures $\mathcal{B}_L^*$ and $\mathcal{B}_U^*$ defined as follows are the smallest and the greatest (with respect to R) solutions of M' that satisfy those conditions.*

$$
\begin{array}{l}
\langle \mathcal{B}_L^0, \mathcal{B}_U^0 \rangle := \langle \mathcal{B}_L, \mathcal{B}_U \rangle. \\
\textit{for all ordinal } \alpha\textit{:} \\
\quad \mathcal{B}_L^{\alpha+1} \in M(\mathcal{B}_L^\alpha|_{vocab(\mathcal{B}_L)\setminus\{S\}} \,||\, \mathcal{L}) \textit{ where } dom(\mathcal{L}) = dom(\mathcal{B}_L) \textit{ and } S^{\mathcal{L}} = R^{\mathcal{B}_L^\alpha}, \\
\quad \mathcal{B}_U^{\alpha+1} \in M(\mathcal{B}_U^\alpha|_{vocab(\mathcal{B}_U)\setminus\{S\}} \,||\, \mathcal{U}) \textit{ where } dom(\mathcal{U}) = dom(\mathcal{B}_U) \textit{ and } S^{\mathcal{U}} = R^{\mathcal{B}_U^\alpha}. \\
\textit{for limit ordinal } \alpha\textit{:} \\
\quad \langle \mathcal{B}_L^\alpha, \mathcal{B}_U^\alpha \rangle := \langle \bigsqcup_{\beta<\alpha} \mathcal{B}_L^\beta, \bigsqcap_{\beta<\alpha} \mathcal{B}_U^\beta \rangle.
\end{array}
\tag{4.10}
$$

Proof: The proof is again a standard Tarski proof. ∎

One of the consequences of Theorem 4.7 is that, now, under certain conditions, we can improve and refine the complexity result of Theorem 4.5. Theorem 4.5 states that the set of problems that can be described in terms of modular systems with feedback and with primitive modules that are decidable in Δ_k^P is exactly the set of problems that can be decided in Σ_{k+1}^P. Here, we can refine this theorem by saying that all the modular systems that use only feedback and composition operators and whose primitive modules are Δ_k^P solvable, total and monotone are Δ_k^P solvable themselves. Theorem 4.8 that follows states this fact:

Theorem 4.8 (Δ_k^P Solvability for Monotone Modular Systems) *Let $M \in \mathrm{MS}(\sigma, \varepsilon)$ be a modular system such that all primitive modules $M' \in \mathrm{MS}(\sigma', \varepsilon')$ of M are Δ_k^P solvable, σ'-total and σ'-τ-ε'-monotone where $\tau \subseteq \sigma$. Then, M is also Δ_k^P solvable.*

Proof: We just iterate the application of all primitive modules on the empty expansion of instance structure $\mathcal{A}$ to obtain the smallest solution to M. Since the interpretation to all vocabulary symbols increases monotonically and because the union operator is not allowed, this process can continue at most polynomially many times. Also, since all modules are Δ_k^P solvable, our procedure is in $P^{\Delta_k^P} = \Delta_k^P$. ∎

Now, let us turn to anti-monotone (negative) loops and consider how they affect the set of possible solutions.

Proposition 4.13 (Anti-Monotonicity and Minimality) *For $\{S\}$-τ-$\{R\}$-anti-monotone modular system M and for modular system $M' := M[S = R]$, we have that when interpretation to τ is fixed, all models of M' are minimal with respect to the interpretations of R.*

Proof: Let $\mathcal{B}_1, \mathcal{B}_2 \in M'$ be such that $B_1|_\tau = B_2|_\tau$ and $R^{\mathcal{B}_1} \sqsubseteq R^{\mathcal{B}_2}$. So, because, in M', R is fed back to S, we have $S^{\mathcal{B}_1} \sqsubseteq S^{\mathcal{B}_2}$. Hence, by $\{S\}$-τ-$\{R\}$-anti-monotonicity of M, we have that $R^{\mathcal{B}_1} \sqsupseteq R^{\mathcal{B}_2}$. Thus, $R^{\mathcal{B}_1} = R^{\mathcal{B}_2}$ and $B_1|_{\tau \cup \{R\}} = B_2|_{\tau \cup \{R\}}$, i.e., there does not exist any two structures in M' which agree on the interpretation to τ but, in one of them, interpretation of R properly extends R's interpretation in the other one. ∎

The minimality of solutions to anti-monotone loops means that these loops may not have a smallest solution. Nevertheless, we are still able to prune the candidate solution space by finding lower and upper bounds for all the solutions to such a loop. Consider the following process for a $(\tau \cup \{S\})$-total and $\{S\}$-τ-$\{R\}$-anti-monotone modular system M where S and R are relational symbols of arity n:

$$\begin{aligned} L_0 = \emptyset, U_0 = [dom(\mathcal{A})]^n, \\ L_{i+1} = R^{M(\mathcal{A} \,||\, \mathcal{U}_i)}, U_{i+1} = R^{M(\mathcal{A} \,||\, \mathcal{L}_i)}, \end{aligned} \tag{4.11}$$

where $dom(\mathcal{L}_i) = dom(\mathcal{U}_i) = dom(\mathcal{A})$, $S^{\mathcal{L}_i} = L_i$, $S^{\mathcal{U}_i} = U_i$ and, for two structures $\mathcal{A}_1$ and $\mathcal{A}_2$ over the same domain but distinct vocabularies, $\mathcal{A}_1||\mathcal{A}_2$ is defined to be the structure over the same domain as $\mathcal{A}_1$ and $\mathcal{A}_2$ and with the same interpretation as them.

Theorem 4.9 (Bounds on Solutions to Anti-Monotone Loops) *For $(\tau \cup \{S\})$-total and $\{S\}$-τ-$\{R\}$-anti-monotone modular system $M \in \mathrm{MS}(\sigma, \varepsilon)$ (where $S \in \sigma$ and $R \in \varepsilon$ are symbols of arity*

n), and for modular system $M' := M[S = R]$ and τ-structure $\mathcal{A}$, the approximation process (4.11) *has a fixpoint $(L^*_{\mathcal{A}}, U^*_{\mathcal{A}})$ such that for all $\mathcal{B} \in M'$ with $\mathcal{B}|_\tau = \mathcal{A}$, we have $L^*_{\mathcal{A}} \sqsubseteq R^{\mathcal{B}}$ and $R^{\mathcal{B}} \sqsubseteq U^*_{\mathcal{A}}$.*

Proof: We prove this for relational symbols. Extending it to function symbols is straightforward. Given τ-structure $\mathcal{A}$, consider the set $S = \{\mathcal{B} \in M' \mid \mathcal{B}|_\tau = \mathcal{A}\}$. We first prove (by induction on i) that, for all i, we have: $L_i \sqsubseteq L_{i+1}$, $U_i \sqsupseteq U_{i+1}$, $L_i \sqsubseteq \bigsqcap_{\mathcal{B}\in S} R^{\mathcal{B}}$, and $U_i \sqsupseteq \bigsqcup_{\mathcal{B}\in S} R^{\mathcal{B}}$.
The base case is easy because L_0 is the empty set and U_0 contains all possible tuples. For the inductive case:

1. By induction hypothesis, $U_i \sqsupseteq U_{i+1}$. So, by anti-monotonicity of M, we have: $L_{i+1} = L^{M(\mathcal{A} \,||\, \mathcal{U}_i)} \sqsubseteq L^{M(\mathcal{A} \,||\, \mathcal{U}_{i+1})} = L_{i+2}$. Similarly, $U_{i+1} \sqsupseteq U_{i+2}$.
2. Again, by induction hypothesis, $U_i \sqsupseteq \bigsqcup_{\mathcal{B}\in S} R^{\mathcal{B}}$. So, for all structures $\mathcal{B} \in S$, we have: $U_i \sqsupseteq R^{\mathcal{B}}$. Therefore, $L_{i+1} = L^{M(\mathcal{A} \,||\, \mathcal{U}_i)} \sqsubseteq R^{\mathcal{B}}$. Thus, $L_{i+1} \sqsubseteq \bigsqcap_{\mathcal{B}\in S} R^{\mathcal{B}}$. Similarly, we also have $U_{i+1} \sqsupseteq \bigsqcup_{\mathcal{B}\in S} R^{\mathcal{B}}$.

So, as $\bigsqcap_{\mathcal{B}\in S} R^{\mathcal{B}} \sqsubseteq \bigsqcup_{\mathcal{B}\in S} R^{\mathcal{B}}$, we have that, for all i, $L_i \sqsubseteq U_i$. Thus, there exists ordinal α where (L_α, U_α) is the fixpoint of the sequence of pairs (L_i, U_i). Denote this pair by $(L^*_{\mathcal{A}}, U^*_{\mathcal{A}})$. Observe that, by above properties, $L^*_{\mathcal{A}} \sqsubseteq R^{\mathcal{B}}$ and $R^{\mathcal{B}} \sqsubseteq U^*_{\mathcal{A}}$ for all $\mathcal{B} \in S$ (as required). ∎

Similar to Theorem 4.6, Theorem 4.9 also prunes the search space by limiting the candidate solutions to only those that are both supersets of the lower bound obtained by the process and subsets of the upper bound obtained by it.

Example 4.27 (Well-Founded Models) *As discussed in Example 4.26, the module $M' := M_1 \rhd M_2$ is $\{S\}$-$\{P\}$-$\{S'\}$-anti-monotone. Thus, by Proposition 4.13, the module M defined in Example 4.7 can only have minimal solutions with respect to symbol S for a fixed input P. Moreover, by Proposition 4.9, we can find lower and upper bounds to all the solutions of module M for a fixed P. Unsurprisingly, these bounds coincide with the well-founded model of the logic program P.*

Theorem 4.9 can also be extended to work with general lower and upper bounds as in the case of Theorem 4.6.

Theorem 4.10 (Bounding Solutions to Negative Feedbacks) *Let M be a $(\tau \cup \{S\})$-total and $\{S\}$-τ-$\{R\}$-anti-monotone modular system and $M' := M[S = R]$. Also, let $\mathcal{B}_L$ and $\mathcal{B}_U$ be two structures such that $\mathcal{B}_L|_{vocab(\mathcal{B}_L)\setminus\{R,S\}} \sqsubseteq \mathcal{B}_U|_{vocab(\mathcal{B}_U)\setminus\{R,S\}}$ and $S^{\mathcal{B}_L} = R^{\mathcal{B}_L} \sqsubseteq R^{\mathcal{B}_U} = S^{\mathcal{B}_U}$. Then, if $\mathcal{B}$ is a solution of M' for which τ is interpreted as in $\mathcal{B}_L$ and R is interpreted in the range defined by $R^{\mathcal{B}}_L$ and $R^{\mathcal{B}}_U$, then interpretation of R in $\mathcal{B}$ also falls within the range of R's interpretation in the*

structures $\mathcal{B}_L^*$ and $\mathcal{B}_U^*$ that are defined as follows.

$$
\begin{array}{l}
\langle \mathcal{B}_L^0, \mathcal{B}_U^0 \rangle := \langle \mathcal{B}_L, \mathcal{B}_U \rangle. \\
\textit{for all ordinal } \alpha\textit{:} \\
\mathcal{B}_L^{\alpha+1} \in M(\mathcal{B}_U^{\alpha}|_{vocab(\mathcal{B}_U)\setminus\{S\}} \,||\, \mathcal{U}) \textit{ where } dom(\mathcal{U}) = dom(\mathcal{B}_U) \textit{ and } S^{\mathcal{U}} = R^{\mathcal{B}_U^{\alpha}}, \\
\mathcal{B}_U^{\alpha+1} \in M(\mathcal{B}_L^{\alpha}|_{vocab(\mathcal{B}_L)\setminus\{S\}} \,||\, \mathcal{L}) \textit{ where } dom(\mathcal{L}) = dom(\mathcal{B}_L) \textit{ and } S^{\mathcal{L}} = R^{\mathcal{B}_L^{\alpha}}. \\
\textit{for limit ordinal } \alpha\textit{:} \\
\langle \mathcal{B}_L^{\alpha}, \mathcal{B}_U^{\alpha} \rangle := \langle \bigsqcup_{\beta<\alpha} \mathcal{B}_L^{\beta}, \bigsqcap_{\beta<\alpha} \mathcal{B}_U^{\beta} \rangle.
\end{array}
\tag{4.12}
$$

Proof: Proof is similar to Theorem 4.9. ∎

Note that Theorem 4.10 just gives some possible bounds on the set of possible structures and that these bounds are not always the optimal bound. Example 4.27 that we discussed before gives an example of when such bounds are not optimal, i.e., while founded atoms in well-founded models are always a subset of all stable models (and thus a subset of the intersection of all stable models), there exist logic programs such as P so that the intersection of all stable models of P properly contains the set of founded atoms in the well-founded model of P. Thus, Example 4.27 concretely shows that negative feedback approximations are not generally the best approximation possible. In fact, in this case, if we are guaranteed to know that negative feedback approximations are optimal (i.e., they coincide with intersection of all solutions) then we can obtain a similar result to Theorem 4.8 to reduce the complexity of solving a modular system.

4.7.2 Extending Lazy Modular Solving Algorithm with Approximation Techniques

In this section, we extend Algorithm 2 using the approximation procedures that we introduced in Section 4.7.1. The extended algorithm prunes the search space of a model by propagating information obtained by these approximation procedures to the solver.

We use procedures (4.10) and (4.12) to obtain a set of increasing and decreasing sequences that bound the set of possible solutions. Using these procedures, we extend Algorithm 2 to Algorithm 3. Equations (4.10) and (4.12) guide our solver towards the right direction and, thus, try to speed up the search process. Algorithm 3, thus, depends on a solver that can deal with propagated literals. Also, in Algorithm 3 we use notation $\neg L$ (for a set L of literals) to denote $\{\neg l \mid l \in L\}$.

Proposition 4.14 (Correctness of Extended Algorithm) *Algorithm 3 is sound and complete for finite structures, i.e., given a modular system M with CCAV oracles, a complete online solver S and a finite instance structure $\mathcal{A}$:*

Data: Similar to Algorithm 2, but with modules' totality, monotonicity and anti-monotonicity properties given
Result: Structure $\mathcal{B}$ 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 the set of advices 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** $\mathcal{B}$;
      else
        foreach *applicable pos. (resp. neg.) feedback* $M_1 := M_2[T = T']$ **do**
          Let $\langle L^*, U^* \rangle$ be the limit of series $\langle L_i, U_i \rangle$ in Eq. 4.10 (resp. Eq. 4.12);
          Propagate $(L^* \backslash T^{+^{\mathcal{B}}}) \cup \neg([dom(\mathcal{A})]^n \backslash (U^* \cup T^{-^{\mathcal{B}}}))$ to S ;
end

Algorithm 3: Lazy Model Expansion with Approximation (Propagation)

1. *If Algorithm 3 returns $\mathcal{B}$, then $\mathcal{B} \in M$,*
2. *If Algorithm 3 returns "*Unsatisfiable*" then none of structures $\mathcal{B} \in M$ expand $\mathcal{A}$.*
3. *Algorithm 3 always terminates.*

Proof: The proof is similar to the proof of Proposition 4.4, but it also depends on the correctness of our approximation procedures from Section 4.7.1. ∎

4.8 Related Work

Many works have been done on modularity in declarative programming, we only review the most relevant ones. The authors of [120] proposed a multi-language framework for constraint modelling. That work was the initial inspiration of our earlier work [187], but we 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 [63]. The authors derive a semantics for modular logic programs by viewing a logic program as a generalized quantifier. The ideas are further generalized in [160] 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 [119] to define modularity for disjunctive logic programs. There are also other approaches to adding modularity to ASP languages [46, 109, 18, 15]. The related approach of ID-Logic is described in [55].

The works mentioned earlier focus on the theory of modularity in declarative languages. There also exist research directions that focus on the practice of modular declarative programming and, in particular, solving. These generally fall into one of the following three categories:

The first category consists of practical modelling languages which incorporate other modelling languages. For example, X-ASP [181] and ASP-PROLOG [66] extend Prolog with ASP, CP techniques are incorporated into ASP solving in [20], [143], and [84]. Also, ESRA [72], ESSENCE [76] and Zinc [48] are CP languages extended with features from other languages. 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 concerned with automatic extraction of independent modules from logic programs, i.e., to break a logic program into layers such that each layer depends only on the modules below that layer. In order to achieve this, different splitting theorems for logic programs have been devised. Lifschitz and Turner [135] showed some syntactic conditions to split a disjunctive logic program Π into two disjunctive logic programs Π_1 and Π_2 such that S is an answer set of Π if and only if $S \cap vocab(\Pi_1)$ is an answer set of Π_1 and S is an answer set of $\Pi_2 \cup (S \cap vocab(\Pi_1))$. Applying this process repeatedly splits a logic program into several levels. Such a splitting process speeds up the computation of answer sets for logic programs by finding a natural and linear representation of the same problem. This process is generalized to other frameworks for non-monotonic reasoning such as default theories [199]. Also, Vennekens and others [207] give an algebraic account of splitting that is applicable to a wide array of different semantics.

The third category is to design methods to solve specific problems that use modules from different communities. The field of "Integrated Methods" [113] is an active field that aims at solving particular challenges by combining techniques from operations research (with techniques such as relaxations and bounding), constraint programming (with techniques such as constraint propagation

and arc-consistency) and artificial intelligence (with its heuristic techniques). Combination of all these different techniques from different fields has led to great success stories of solving specific problems [114, 112, 73]. Integrated methods are similar to our research because we both believe in the importance of problem-specific solutions. This is emphasized in our work through oracles that are specific to modules (in contrast to allowing only generic oracles). However, despite this similarity, our research is fundamentally different from that of integrated methods because we are primarily concerned with the development of a unifying model-theoretic approach towards modular systems. In other words, combination of different solving techniques from different fields is just a consequence of our model-theoretic development of modular systems and not the main driving force behind it. Moreover, our way of developing modular systems allow us to study properties about modular systems that would have been very hard to study otherwise. Such properties include (but are not restricted to) studying the computational complexity of performing different logical tasks (such as model checking, model expansion, satisfiability, brave/cautious reasoning) for modular systems, studying expressiveness of modular systems for different classes of structures (such as finite structures, embedded structures, etc.), studying abstract approximation techniques, etc.

A different but related approach to modularity is that of multi-context systems (MCS). A multi-context system is a collection of contexts that are connected together through bridge rule. Each context represents a knowledge base as well as the semantics under which the knowledge base is interpreted. Bridge rules act as carriers of information between contexts. They are similar to logic programming rules and assert the truth of the belief in the head of a rule whenever the body of that rule is satisfied. The semantics of a multi-context system is a collection of belief sets for different context that form a fixpoint of the multi-cotext system. In [25], 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 [14, 23, 61, 60]. The main difference between our approach and the approach of multi-context systems is due to our different intended applications: while multi-context systems are designed to specifically address integration of declarative knowledge bases, our approach aims at addressing a broader set of applications with primitive modules coming from both declarative and non-declarative settings. Such different intended use-cases also cause another apparent difference between our approach and the MCS approach: in multi-context systems, because of the essential role that knowledge bases play, there is an inherent tendency to model both the syntax and semantics of knowledge bases. This is in contrast to our approach that abstracts away both the syntax and internal semantics of a primitive module. Instead, we focus on what a primitive modules represents.

Despite the above-mentioned differences between our approach and the MCS approach, we believe that there are more similarities between these two approaches than there are differences. The most important similarity is due to the similarity in the concepts that these two approaches address. In this chapter, we focused on solving a modular system. Similarly, in [14], the authors address the similar issue of finding equilibria for multi-context systems. Another example that has been addressed in the area of multi-context systems but has yet to be addressed for our framework is the concept of diagnosis. Similar to MCS's, modular systems are also combinations of (usually) highly complex primitive modules. Thus, the combined modular system represents an even more complex entity. Therefore, to design a correct modular system, one needs some tools to help with checking combined system's correctness. This issue is addressed in [23, 61, 60]. Another issue that has been partially developed for modular systems but has yet to be developed for MCS's is the concept of approximations. In modular system, approximations are used to rapidly approximate solutions of some feedbacks in a modular system [187]. A similar concept can be developed for approximating equilibria in multi-context systems. To summarize, we believe that the fields of modular systems and multi-context systems share a great amount of mutual interest and that the connection between these two directions should be studied formally.

4.9 Conclusion

In this chapter, we introduced a framework of modular systems and showed how it can be used for modular declarative modelling. The framework allows primitive modules to be combined through serial combination of modules (i.e, composition), parallel combination of modules (i.e., union), abstraction operation (i.e., projection) and feedbacks. These operations may be viewed as a counterpart of the well-known relational algebra operations, but on sets of structures rather than on relational tables. We defined two semantics for algebraic expressions representing modular systems – a model-theoretic semantics and a semantics based on fixpoints of operators associated with modular systems. We showed that these semantics are equivalent. Both of these semantics are defined so that they are independent of how a module is specified, which can be done for example through a decision procedure or through an axiomatization in some logic.

Moreover, we also introduced an operational view on (and structural operational semantics for) modular systems and discussed the transient states of modular systems. Transient states can be used to reason about modular systems even when the modular system does not have a fixpoint. A detailed study of transient states of modular systems is a subject of future research.

We also discussed the complexity and the expressiveness of modular systems framework in the presence/absence of some of the operations used for combining modular systems. We showed that both the operations of union and feedback are capable of creating a jump in the expressiveness of modular system frameworks from Δ_k^P to Σ_k^P levels of the Polynomial time hierarchy.

An interesting application of the algebra of modular systems is that, for example, ASP programs, viewed as individual modules, can be combined both disjunctively or using feedbacks. Although useful from both the convenience of knowledge representation and extended formal expressiveness point of view, these combinations were not possible in the context of ASP programs themselves. The detailed study of how modular specification can affect answer set programming is a subject of future research.

Our language-independent view on modular systems enabled us to develop an algorithmic schema for solving modular systems abstractly (i.e., without concerning what language each module uses internally). Our algorithm is designed to solve the model expansion task (i.e., to search for a solution) for modular systems that represent combinatorial search problems. We evaluated our algorithm through abstractly modelling several existing systems, i.e., DPLL(T), ILP, and ASP+CP. We demonstrated that, in the context of model expansion, our algorithm generalizes the work of these solvers. We also showed that several different procedures that are used in different communities for information propagation (such as unit propagation, well-founded model computation and arc-consistency checks) can be unified as a single concept of an *advice* in our algorithm. Similarly, we argued that our *valid acceptance procedures* also generalize similar concepts in different communities.

We extended our algorithmic schema for solving modular systems with an approximation mechanism that uses the operational view on modular systems. The algorithm extends a partial structure with positive/negative facts about the possible solutions of the modular system. Our approximation mechanism is applicable for the case of positive/negative loops, i.e., modular systems that are formed by creating a feedback from a symbol such as S to a symbol such as R and knowing that S monotonically increases/decreases by increasing R. We also investigated some cases where such monotonic properties guarantee the existence of a better (i.e., computationally less complex) procedure for solving modular systems. A more detailed analysis of other cases than can be used for achieving solutions is an interesting future direction.

Chapter 5

Modular Systems with Supports

5.1 Motivation

In Chapter 4, we showed that the model-theoretical semantics of modular systems coincides with a fixpoint semantics for the operational view on modular systems. The fixpoint semantics for modular systems does not differentiate between different types of fixpoints. However, in practice, we are usually interested only in "justifiable" fixpoints. The following example showcases our meaning of a justifiable fixpoint:

Example 5.1 (Shopping) *Consider John who needs to buy a set of matching clothes for a formal event he would be attending. He has to choose his type of clothing (e.g. between a tie and a bowtie), the color of the suit he is going to buy and the matching shirt. For example, if he buys a dark suit, he will also have to buy a white shirt. An intended solution for this problem is a solution in which, after shopping, John has a fully matching set of clothes.*

In a fixpoint semantics such as the semantics of modular systems, one of the non-intended models for this problem is that John would end up buying every possible set of matching clothes he can find in a shopping mall. Although this solution satisfies his goal of having a set of matching clothes to wear, it is an unreasonable solution because so much extravagance is unjustified: he would have reached his goal with just one set of those clothes as well.

The problem in Example 5.1 is that fixpoint semantics does not provide any means for accompanying actions with their justifications. Under such a semantics, there is no easy way to guarantee that John of Example 5.1 buys only one set of matching clothes. This type of problem happens in

many other situations as well. One of the examples that is frequently referenced in the literature is the connectivity example that follows.

Example 5.2 (Connectivity) *You are given a set V of vertices and a set E of edges in a graph and you are asked to find all pairs (u, v) of vertices such that u and v are connected through edges in E. We know that this problem is polytime computable but if we try to formalize it as a fixpoint compoutation, we would have something as follows: find the fixpoint of function $C_E(R) := E \cup \{(u, u) \mid u \in V\} \cup \{(u, v) \mid \exists w\ ((u, w) \in E \wedge (w, v) \in R)\}$. However, we know that the only interesting fixpoint for this function is its least fixpoint (which characterizes the solution to our connectivity problem). Moreover, we also know that the connectivity problem cannot be characterized as the fixpoint of any function that uses only first order sentences for constructing sets.*

Example 5.2 showcases a situation where not only is the notion of justification needed to obtain a reasonable fixpoint semantics but also a situation where justifications can provide meaningful information about connectivity of two nodes. The following example shows how justifications can be used to identify paths between connected nodes.

Example 5.3 (Connectivity: Justifications as Paths) *Consider the connectivity problem of Example 5.2. Assume that, for each pair $(u, v) \in R$ of connected nodes, we denote the justification for connectivity of (u, v) using $j_R(u, v)$ which satisfies the following conditions:*

if $u = v$ then $j_R(u, v) = \{\}$,
if $u \neq v$ and $(u, v) \in E$ then $j_R(u, v) = \{v\}$, and,
otherwise, $j_R(u, v) = \{w\}$ for some w such that $(u, w) \in E$ and $(w, v) \in R$.

Note that, for all fixpoints R of C_E (i.e., if $R = C_E(R)$), at least one function $j_R(u, v)$ exists that satisfies the required conditions because, if R is a fixpoint of C_E and $(u, v) \in R$ then at least one of the conditions (1) $u = v$, (2) $(u, v) \in E$, or, (3) $\exists w\ ((u, w) \in E \wedge (w, v) \in R)$ is true. Moreover, if R is the least fixpoint of equation $R = C_E(R)$, we can define function $j_R(u, v)$ such that $j_R(u, v)$ contains the last node visited in a path from u to v.

Such a justification function carries a very useful piece of information: the path that starts at u and ends at v. This path can be computed as follows:

$$path_j(u, v) := \begin{cases} [v] & \text{if } u = v, \\ (u : path_j(w, v)) & \text{if } u \neq v \text{ and } j(u, v) = \{w\}. \end{cases}$$

where $[x]$ and $(x : xs)$ respectively denote a list containing only one element x, and a list that starts with element x and is followed by list xs (as is customary in functional programming languages).

Examples 5.1 and 5.2 define two cases when we are interested only in justifiable fixpoints and not in all fixpoints. Moreover, Example 5.3 shows how certain justifications can contain useful information about a model. It remains to see how exactly such useful justifications can be characterized. Following example demonstrates the essential property that characterizes useful justifications in the context of connectivity problem.

Example 5.4 (Connectivity: Non-circular Justifications) *Again, consider the problem of connectivity as in Example 5.2. Also, let R be a fixpoint of equation $R = C_E(R)$ and j_R be a justification function for R that satisfies the conditions of Example 5.3. Moreover, let us take our measure of usefulness for a justification j_R to be the finitude of lists created by function $path_j$. That is, for the connectivity problem, we say that a justification function j_R is useful if, for all $(u, v) \in R$, the list created by $path_j(u, v)$ is finite.*

Note that, the finitude of a list created by $path_j(u, v)$ is indeed an important property because if $path_j(u, v)$ is finite then it represents a path that starts at u and ends at v. This is because the conditions on j_R guarantees that, for all $n < length(path_j(u, v))$, the n-th element and the $(n + 1)$-th element of the list $path_j(u, v)$ are connected to each other through an edge. Therefore, if $(u, v) \in R$ and $path_j(u, v)$ is a finite list then $path_j(u, v)$ witnesses the connectivity of u to v. Also, since all fixpoints of equation $R = C_E(R)$ include all the connected pairs of vertices, the finitude of $path_j(u, v)$ for all $(u, v) \in R$ witnesses the minimality of R, i.e., R includes all and only the connected pairs of vertices.

So, our question now is how we can characterize the finitude of list $path_j(u, v)$ for all $(u, v) \in R$. For finite graphs, this property can be easily guaranted: $path_j(u, v)$ is finite if and only if $path_j(u, v)$ contains each vertex at most once. This is because (1) $path_j(u, v)$ would be finite if it contains each vertex at most once (there are only finitely many vertices in a finite graph), and, (2) if $path_j(u, v)$ contains a vertex w twice it means that $path_j(w, v)$ depends on itself and, thus, the construction of $path_j(w, v)$ will not finish in a finite number of steps and, so, $path_j(u, v)$ will be infinite. Hence, in finite graphs, $path_j(u, v)$ is finite for all $(u, v) \in R$ if and only if the definition of $path_j(u, v)$ is non-circular for all $(u, v) \in R$.

Henceforth, justification function j is useful if and only if $j(u, v)$ does not depend on u (either directly or indirectly) for all $(u, v) \in R$.

Example 5.4 shows how the non-circularity of justifications help us characterize useful justifications for connectivity problem in finite graphs[1]. Interestingly, the condition of non-circularity characterizes many other useful justification functions as well. For example, both the semantics of Horn formulas and stable model semantics can be characterized as sets of "reasonable" fixpoints with non-circular (and well-founded) justifications [134]. Example below demonstrates this correspondence in the case of stable model semantics for normal propositional logic programs.

Example 5.5 (Stable Model Semantics: Well-justified) *Consider a normal propositional logic program P, i.e., P is a set of rules of form $h \leftarrow b_1, \cdots, b_k, \textbf{not}\, b_{k+1}, \cdots, \textbf{not}\, b_n$ where h and $b_1, \cdots, b_n$ are propositional atoms. Then, by [134], we know that for all sets S of propositional atoms, S is a stable model of P if and only if a well-founded ordering $<$ on S exists so that, for all $a \in S$, we have a rule $a \leftarrow b_1, \cdots, b_k, \textbf{not}\, b_{k+1}, \cdots, b_n$ in P such that:*

$$\begin{array}{l} \textit{for all } 1 \leq i \leq k, \textit{ we have } b_i \in S \textit{ and } b_i < a, \textit{ and,} \\ \textit{for all } k+1 \leq i \leq n, \textit{ we have } b_i \notin S. \end{array}$$

Together, Examples 5.1 – 5.5 show the importance of non-circular and well-founded justifications in characterizing intended models of a system. In this chapter, we want to augment modular systems with a similar notion of justifications. As we show in this chapter, adding justifications to the semantics of modular system extends the expressiveness of our modular system framework.

Contributions

The following summarizes our contributions in this chapter.

Justification and Support for Models of Primitive Modules

We augment primitive models of a modular system with the notions of justification and support functions for their models. In essence, each justification function justifies one of the models of a primitive module. Also, support functions assign possible justification functions to a model, i.e., a single model might be associated with several different justification functions.

[1]For infinite graphs, useful justifications can be characterized as those justifications that are both non-circular and well-founded. However, since the proof is more involved and is not directly related to our discussion in this chapter, we do not discuss this more general case in this chapter.

Extending Support to Combined Modular Systems

We recursively define meaningful support (and justification) functions for combinations of modular systems by combining the support functions of combined modules. We show the usefulness of our definitions through several natural examples.

Defining Supported Model Semantics for Modular Systems

We define supported model semantics of modular systems to accept exactly those models of a modular system that are supported. We show that this definition captures the idea of non-circular and well-founded justifications and that many non-trivial semantics can readily be defined in terms of supported semantics for modular systems. We also show that supported model semantics generalizes the previous model-theoretical semantics for modular systems.

Expressing Equilibria of Multi-context Systems using Supported Model Semantics

We show that, the class of multi-context systems with structural belief sets, can be naturally translated into a modular system under supported model semantics. We show that our translation is robust enough to be extended to multi-context systems with variables in their bridge rules.

Expressing Grounded Equilibria of Multi-context Systems using Supported Model Semantics

Similar to the previous case, we also translate multi-context systems under grounded equilibrium semantics to modular systems under supported model semantics. Interestingly, the translation we use for both these multi-context systems is similar to each other and the only difference between these two translations is the justification functions of the primitive modules in these two translations. Therefore, supported model semantics effectively generalizes the two different semantics of normal and grounded equilibria for multi-context systems.

5.2 Background

Atoms in Structures

Recall that a structure $\mathcal{A}$ is a domain (a set of abstract elements denoted by $dom(\mathcal{A})$) plus an interpretation of its vocabulary symbols (denoted by $vocab(\mathcal{A})$). While this definition satisfactorily

represents structures, in this chapter, we represent structures in a different form that is best suited to the goals of this chapter.

Let $\mathcal{A}$ be a structure and let S_r and S_f be, respectively, the set of relational and functional vocabulary symbols of $\mathcal{A}$ (constants are zero-ary functions). Then, we represent $\mathcal{A}$ by its domain plus the set of its truth assignments as follows:

1. For each n-ary relational vocabulary symbol $R \in S_r$ and for each n-ary tuple $t \in R^{\mathcal{A}}$, the truth assignments of $\mathcal{A}$ include atoms of form R_t.
2. For each n-ary functional vocabulary symbol $f \in S_f$ and for each n-ary tuple $t \in [dom(\mathcal{A})]^n$, the truth assignments of $\mathcal{A}$ include atoms of form $f_{t \mapsto a}$ where $a \in dom(\mathcal{A})$ is such that $f^{\mathcal{A}}(t) = a$.
3. Truth of assignments of $\mathcal{A}$ does not contain anything else.

In other words, the set of $\mathcal{A}$'s truth assignments is the following set:

$$\{R_t \mid R \in vocab(\mathcal{A}), R \text{ is a relational symbol, and } t \in R^{\mathcal{A}}\} \cup$$
$$\{f_{t \mapsto a} \mid t \in [dom(\mathcal{A})]^n, f \in vocab(\mathcal{A}), f \text{ is a functional symbol, and } f^{\mathcal{A}}(t) = a\}.$$

Moreover, although tuples are normally encapsulated in angled brackets $\langle . \rangle$, for unary tuples, we sometimes drop these brackets and write R_a and $f_{a \mapsto b}$ instead of, respectively, $R_{\langle a \rangle}$ and $f_{\langle a \rangle \mapsto b}$. Also, for zero-ary vocabulary symbols, we may use R or f_a to respectively denote $R_{\langle\rangle}$ or $f_{\langle\rangle \mapsto a}$. Furthermore, we use the term *atom* (or *true atom*) of a structure $\mathcal{A}$ to denote a member of $\mathcal{A}$'s truth assignments. Also, we denote truth assignments of $\mathcal{A}$ by $at(\mathcal{A})$ (read atoms of $\mathcal{A}$).

5.3 Modular Systems Extended

This section generalizes modular systems of Chapter 4 with supported supported semantics. Informally speaking, supported semantics augments each model with some possible justification functions for that model. While the model itself carries the membership information for tuples, the justification functions carry some information in addition to these membership information: *they partially reason about why each tuple is present in a model*.

Recall that one of the great properties of modular systems is their language-independence that was achieved through using a model-theoretic semantics. Therefore, in this section, we also define supported semantics model-theoretically to preserve the language-independence property. We begin our study by defining justifications of models in a modular system.

Definition 5.1 (Justification for Models) *Let $M \in \mathrm{MS}(\sigma, \varepsilon)$ be a modular system and let $\mathcal{B} \in M$ be a model of M. Then, a function $j : at(\mathcal{B}|_\varepsilon) \mapsto \mathcal{P}(at(\mathcal{B}))$ that maps each true atom of $\mathcal{B}$'s expansion vocabulary to a subset of $\mathcal{B}$'s true atoms is called a* justification function *if and only if a well-founded ordering $<$ on $at(\mathcal{B}|_\varepsilon)$ exists such that:*

$$\textit{for all } R_t \in at(\mathcal{B}|_\varepsilon) \textit{ and for all } S_{t'} \in (j(R_t) \cap at(\mathcal{B}|_\varepsilon)) \textit{ we have } S_{t'} < R_t.$$

Note that Definition 5.1 disallows circular justifications as desired and in accordance with motivations in Section 5.1. This is because, if something depends on itself, by transitivity of ordering relations, we should have $R_t < R_t$ (for some $R_t \in at(\mathcal{B}|_\varepsilon)$) which is impossible in all orderings. Now, let us apply this concept to our example of connectivity in graphs.

Example 5.6 (Connectivity Module) *Consider modular system $M_C \in \mathrm{MS}(\{E\}, \{R\})$ that computes connected pairs R of vertices from the set E of a graph's edges. Also, let $\mathcal{B} \in M_C$ be a model of M_C and define sets S^n (for $n \in \mathbb{N}$) as follows:*

$$S^0 := \{(u, u) \mid u \textit{ is a vertex}\},$$
$$S^{n+1} := S^n \cup \{(u, v) \mid \textit{vertex } w \textit{ exists s.t. } (u, w) \in E^{\mathcal{B}} \textit{ and } (w, v) \in S^n\}.$$

By definition of connectivity, we know that $R^{\mathcal{B}} := \bigcup_{n \in \mathbb{N}} S^n$. Now, define $rank(u, v)$ to be the smallest n such that $(u, v) \in S^n$. Then, a function j is a justification function for $\mathcal{B}$ if it satisfies the following:

$$j(R_{\langle u,v \rangle}) := \begin{cases} \{\} & \textit{if } rank(u, v) = 0, \\ \{E_{\langle u,w \rangle}, R_{\langle w,v \rangle}\} & \textit{if } (u, w) \in E^{\mathcal{B}} \textit{ and } rank(u, v) > rank(w, v). \end{cases}$$

Note that, in Definition 5.1 and, thus, also in Example 5.6, a justification function is defined only on true atoms of output vocabulary, i.e., true atoms of $\mathcal{B}|_\varepsilon$. This is because each module is only responsible for what it generates (i.e., the outputs of the module) and not what it is given (i.e., module's inputs) and, thus, the only reasonable justification one can expect from a module is justification of its outputs. For example, in the connectivity problem, it is reasonable to ask M_C to justify the truth of $R(a, b)$ (because $R(a, b)$ is generated by M_C) but it is unreasonable to ask M_C to justify the existence of an edge between vertices a and b (because M_C only receives edges in its input). The justification for truth of $E(a, b)$ should be requested from the module that has generated interpretation of E or the user of a modular system that has given E as an input.

Informally speaking, justification functions justify true atoms of a structure based on its other (better founded) true atoms. For example, if we have $j(R_a) = \{S_a, S_b\}$, it means that the presence

of tuple $\langle a \rangle$ in interpretation of $R^{\mathcal{B}}$ is justified by S_a and S_b. However, justifications usually refer to incomplete conditionals and should not be misunderstood with logical implication. That is, the fact that R_a in $\mathcal{B}$ is justified by S_a and S_b does not logically imply that whenever S_a and S_b are true atoms of a model then R_a is also a true atom of that model. For example, M might have another model $\mathcal{B}'$ such that $S_a, S_b \in at(\mathcal{B}')$ but $R_a \notin at(\mathcal{B}')$. So, a true atom being justified by a set of other true atoms should not be mistaken with a true atom being implied by that other set of true atoms. Let us now define support functions for modular systems.

Definition 5.2 (Support Functions for Modular Systems) *Let $M \in \mathrm{MS}(\sigma, \varepsilon)$ be a modular system. A* support function *for modular system M, denoted by Sup_M is a function that associates each model $\mathcal{B}$ of M with a set of justification functions for $\mathcal{B}$, i.e., we have:*

$$\textit{if } \mathcal{B} \in M \textit{ and } j \in Sup_M(\mathcal{B}) \textit{ then } j \textit{ is a justification function for } \mathcal{B}.$$

Based on Definition 5.2, $Sup_M(\mathcal{B})$ is a collection of different possible justifications for structure $\mathcal{B}$. The following example shows how to define a support function for our connectivity module.

Example 5.7 (Connectivity Module's Support Function) *Consider module M_C from Example 5.6. For model $\mathcal{B} \in M_C$, define $Sup_{M_C}(\mathcal{B})$ to be the set of all justification functions for structure $\mathcal{B}$ that satisfy the conditions of Example 5.6.*

Note that, based on Definition 5.2, some model $\mathcal{B}$ of a modular system M might be unjustified, i.e., $Sup_M(\mathcal{B}) := \{\}$. Moreover, as we showed in Examples 5.1 – 5.3, using non-circular and well-founded justifications, we can model many problems more naturally. Therefore, we define supported model semantics as follows.

Definition 5.3 (Supported Model Semantics for Modular Systems) *Let $M \in \mathrm{MS}(\sigma, \varepsilon$ be a modular system, Sup_M be a support function for M and $\mathcal{B}$ be a $(\sigma \cup \varepsilon)$-structure. Then, $\mathcal{B}$* is a supported model of *M if (1) $\mathcal{B} \in M$, i.e., $\mathcal{B}$ is a model of M, and (2) $Sup_M(\mathcal{B}) \neq \emptyset$, i.e., $\mathcal{B}$ is supported. Also,* supported semantics of modular systems *is defined to be a semantics for modular systems whose intended models are exactly the set of supported models of M. Moreover, we use $Sup[M] := \{\mathcal{B} \mid \mathcal{B} \in M \textit{ and } Sup_M(\mathcal{B}) \neq \emptyset\}$ denotes the set of* supported models of M.

Definition 6.1 distinguishes supported models simply as models with at least one possible justification. Next, we want to specify how support functions of complex modules can be obtained using support functions of their constituents, e.g., how $Sup_{M_1 \rhd M_2}$ is defined in terms of Sup_{M_1} and Sup_{M_2}. Definitions 5.4–5.7 specify how support functions are combined together.

Definition 5.4 (Support of Composition $M_1 \rhd M_2$) *Let $M := M_1 \rhd M_2$ be a well-formed modular system and let Sup_1 and Sup_2 respectively denote support functions of modular systems M_1 and M_2. Then, support function Sup_M for composition of M_1 and M_2 is defined as follows. For each model $\mathcal{B} \in M$, we define $Sup_M(\mathcal{B})$ to contain exactly those justification functions $j : at(\mathcal{B}|_{\varepsilon_M}) \mapsto 2^{at(\mathcal{B})}$ that are obtained by combining some $j_1 \in Sup_1(\mathcal{B}|_{vocab(M_1)})$ and some $j_2 \in Sup_2(\mathcal{B}|_{vocab(M_2)})$ as below:*

$$j(R_t) := \begin{cases} j_1(R_t) & \text{if } R \in \varepsilon_{M_1}, \\ j_2(R_t) & \text{if } R \in \varepsilon_{M_2}. \end{cases}$$

More intuitively, every justification function $j \in Sup_{M_1 \rhd M_2}(\mathcal{B})$ behaves as a justification function of M_1 when applied on output vocabulary symbols of M_1 and as some other justification of M_2 when applied on ourput vocabulary symbols of M_2. Proposition below states that the function defined by Definition 5.4 is indeed a support function according to Definition 5.2.

Proposition 5.1 *Let $M := M_1 \rhd M_2$ be a well-formed modular system. Then, Sup_M as in Definition 5.4 is well-defined.*

Proof: First, since M is well-formed, outputs of M_1 and M_2 do not interfere, i.e., $\varepsilon_{M_1} \cap \varepsilon_{M_2} = \emptyset$. Therefore, the two cases in Definition 5.4 are mutually exclusive. Also, since $\varepsilon_M = \varepsilon_{M_1} \cup \varepsilon_{M_2}$, the two cases in Definition 5.4 cover all possible cases.
Second, every $j \in Sup_M(\mathcal{B})$ defines a non-circular and well-justified justification function. To prove this, let $j_i \in Sup_i(\mathcal{B}|_{vocab(M_i)})$ (for $i \in \{1, 2\}$) be the two justifications functions that are combined to form function j. Also, let $<_1$ and $<_2$ be two well-founded orderings that witness non-circularity of j_1 and j_2 respectively. Then, it is easy to check that well-founded ordering $<$ defined below witnesses the non-circularity of justification function j. For $R_t, S_{t'} \in at(\mathcal{B}|_{\varepsilon_M})$ we have:

$$R_t < S_{t'} \Leftrightarrow \text{ one of } \left\{ \begin{array}{l} R, S \in \varepsilon_{M_1} \text{ and } R_t <_1 S_{t'}, \\ R, S \in \varepsilon_{M_2} \text{ and } R_t <_2 S_{t'}, or, \\ R \in \varepsilon_{M_1} \text{ and } S \in \varepsilon_{M_2} \end{array} \right\} \text{ holds.}$$

■

Let us now define how support functions change under the projection operator in modular systems. This is much more involved than the previous case because projection hides some of the true atoms that might have been used to justify other atoms. Therefore, in order to find a justification function after a projection, we need a process of "unwinding" that we define below.

Definition 5.5 (τ-Unwinding) *Let $M \in \mathrm{MS}(\sigma, \varepsilon)$ be a modular system, $\mathcal{B} \in M$ be a model of M, j be a justification function for $\mathcal{B}$, and let $\tau \subseteq \varepsilon$ be a subset of output vocabulary symbols. Then, the* result of unwinding j according to τ *is the limit of function series f^n (for $n \in \mathbb{N}$) defined below. All functions f^n map $at(\mathcal{B}|_\varepsilon)$ to $2^{at(\mathcal{B})}$:*

$$f^0(R_t) = j(R_t),$$
$$f^{n+1}(R_t) = \{S_{t'} \mid S_{t'} \in f^n(R_t) \text{ and } S \notin \tau\} \cup \bigcup\{f^n(S_{t'}) \mid S_{t'} \in f^n(R_t) \text{ and } S \in \tau\}.$$

Note that Definition 5.5 is well-defining because the function series f^n always has a limit. This is because, by definition, justification functions are well-founded and non-circular. Therefore, there is no infinite descending chain of true atoms and, hence, for each atom $R_t \in \mathcal{B}|_{(\varepsilon \setminus \tau)}$, there exists a natural number n such that, for all $n' > n$, $f^{n'}(R_t) = f^n(R_t)$. Using the notion of unwinding, we can define the support functions of the projection operator as follows.

Definition 5.6 (Support of Projection) *Let $M' := \pi_\tau(M)$ be a well-formed modular system, $\mathcal{B}' \in M'$ be a model of M', and Sup_M be a support function for M. Then, $Sup_{M'}(\mathcal{B}')$ is the set of functions $j' : at(\mathcal{B}'|_{\varepsilon_{M'}}) \mapsto at(\mathcal{B}')$ for which structure $\mathcal{B}$ and function j exist such that (1) $\mathcal{B} \in M$, (2) $\mathcal{B}|_\tau = \mathcal{B}'$, (3) $j \in Sup_M(\mathcal{B})$, and, (4) unwinding j according to vocabulary $(vocab(M) \setminus \tau)$ produces a function f such that $j'(R_t) = f(R_t)$ for all $R_t \in at(\mathcal{B}'|_{\varepsilon_{M'}})$.*

Informally speaking, Definition 5.6 says that a model $\mathcal{B}'$ of M' is justified if $\mathcal{B}'$ can be expanded to a model $\mathcal{B}$ of M (the underlying not-projected modular system) such that justifications of atoms in $at(\mathcal{B}|_\varepsilon)$ does not depend on the choice of atoms in $at(\mathcal{B}') \setminus at(\mathcal{B})$.

The following definition shows how support functions are defined after applying feedback operations. Remember that each feedback operation changes one input vocabulary symbol to an output vocabulary symbol. Therefore, the new justification function should also justify the atoms of the new output vocabulary symbol.

Definition 5.7 (Support of Feedback $M[P = Q]$) *Let $M' := M[P = Q]$ be a well-formed modular system with $P \in \sigma_M$ and $Q \in \varepsilon_M$. Then, $Sup_{M'}(\mathcal{B})$ is the set of all functions $j' : at(\mathcal{B}|_{(\varepsilon_M \cup \{P\})}) \mapsto 2^{at(\mathcal{B})}$ such that:*

(1) $j'(P_t) = \{Q_t\}$,

(2) justification function $j \in Sup_M(\mathcal{B})$ exists so that $j'(R_t) = j(R_t)$ for all $R_t \in at(\mathcal{B}|_{\varepsilon_M})$, and,

(3) j' is well-founded and non-circular, i.e., well-founded ordering $<$ on $at(\mathcal{B}|_{(\varepsilon_M \cup \{P\})})$ exists such that, for all $R_t, S_{t'} \in at(\mathcal{B}|_{(\varepsilon_M \cup \{P\})})$, if $S_{t'} \in j'(R_t)$ then $S_{t'} < R_t$.

Informally speaking, a feedback operator is saying that all atoms of $\mathcal{B}|_{\varepsilon_M}$ are justified as before and the atoms of the new output vocabulary symbol P are simply justified through the atoms of Q (because P's interpretation is exactly Q's interpretation). However, feedbacks can generate self-justifying loops because atom Q_t might have been justified (either directly or indirectly) by P_t in M. In such a situation, adding feedback creates circular justifications because Q_t is justified by P_t as before, and P_t is now justified by Q_t. In order to disallow such circular justifications, the third condition of Definition 5.7 is added to guarantee that j' is a well-founded and non-circular justification function.

Definition 5.8 (Support of Union) *Let $M := M_1 \cup M_2$ be a well-formed modular system, $\mathcal{B} \in M$ be a model of M, and Sup_1, Sup_2 be support functions for M_1 and M_2 respectively. Then $Sup_M(\mathcal{B})$ is the set of justification functions $j : at(\mathcal{B}|_{\varepsilon_M}) \mapsto 2^{at(\mathcal{B})}$ such that the following conditions is true for either $i = 1$ or $i = 2$: $\mathcal{B}|_{vocab(M_i)} \in M_i$ and justification function $j' \in Sup_i(\mathcal{B}|_{vocab(M_i)})$ exists such that (a) $j(R_t) = j'(R_t)$ for all $R_t \in at(\mathcal{B}|_{\varepsilon_{M_i}})$, and, (b) $j(R_t) = \{\}$ for all $R_t \in (at(\mathcal{B}|_{\varepsilon_M}) \setminus at(\mathcal{B}|_{\varepsilon_{M_i}}))$.*

Now that we have defined all support functions for all operations of a modular system, we can state some of the properties of supported model semantics for modular systems. The first property that follows states that supported model semantics generalizes the model-theoretical semantics. We obtain this result through using a naive justification function known as *empty justification* that is defined below.

Definition 5.9 (Empty Justification) *Let $M \in \mathrm{MS}(\sigma, \varepsilon)$ be a modular system and $\mathcal{B} \in M$ be a model of M. Then, the justification function $e : at(\mathcal{B}|_{\varepsilon}) \mapsto 2^{at(\mathcal{B})}$ where $e(R_t) := \{\}$ for all $R_t \in at(\mathcal{B}|_{\varepsilon})$ is called* the empty justification for $\mathcal{B}$. *Also, the support function $Sup_\emptyset$ of M that associates all $\mathcal{B} \in M$ to singular set $\{e\}$ is called the* empty support function of M.

Theorem 5.1 *Let $M \in \mathrm{MS}(\sigma, \varepsilon)$ be a well-formed modular system so that all primitive modules of M are supported by the empty support function $Sup_\emptyset$. Then, for all $(\sigma \cup \varepsilon)$-structures $\mathcal{B}$, we have that $\mathcal{B} \in M$ if and only if $\mathcal{B}$ is a supported model of M.*

Proof: ($\Leftarrow$) If $\mathcal{B}$ is a supported model of M, by definition, it should also be a model of M.
($\Rightarrow$) A simple induction on the structure of M shows that for all subsystems M' of M, $Sup_{M'}(\mathcal{B}') = \{e\}$ (for all $\mathcal{B}' \in M'$). Therefore, taking $M' := M$ and $\mathcal{B}' := \mathcal{B}$, we have empty justification

function $e \in Sup_M(\mathcal{B})$. Therefore, $\mathcal{B}$ is a supported model of M because it is a model of M by assumption and it is justified. ∎

Theorem 5.1 states that supported model semantics naturally generalizes the model-theoretic semantics for modular systems. Moreover, it states that empty justification functions define the essence of the model-theoretic semantics for modular systems. That is, when justification functions do not provide any extra information about possible reasons for believing in something, every model is as reasonable as every other model. Of course, when we use some non-trivial justification functions except the empty justification function, some models (i.e., supported models) become more reasonable than other models (i.e., non-supported models).

In the next section, we show that supported model semantics for modular systems is expressive enough to include both the equilibrium semantics of multi-context systems and the grounded equilibrium semantics of multi-context systems (when contexts use structural belief sets).

5.4 From Multi-Context to Modular Systems

This section establishes a formal connection between the expressiveness of multi-context systems and that of modular systems. We show that, in the presence of some very basic modules, multi-context systems can be encoded as modular systems. In order to do so, we give a natural translation T from MCSs to modular systems such that changing the support functions of primitive modules of the resulting modular system, we can characterize both the equilibrium semantics and the grounded equilibrium semantics of MCSs in terms of supported models of the resulting modular system. Therefore, we prove that supported model semantics of modular systems generalizes and inifies both types of equilibrium semantics (grounded or not) of multi-context systems.

5.4.1 Encoding MCSs and Equilibria

We first translate multi-context systems to modular systems and then talk about two possible choices as the support functions of the primitive modules used in our translation. As modular systems do not support bridge rules, we encode bridge rules using modules that perform relational algebraic operations. In our translation, we assume that each such operation is modeled by a primitive module. We use $M_{\bowtie}$ for performing relational algebraic join, $M_{\cup}$ for performing relational algebraic union, M_{Π} for performing relational algebraic projection, and M_{Compl} for performing relational algebraic complementation. We put a lot of emphasize on the term "relational algebraic" to make sure that the reader understands the difference between, e.g., primitive module $M_{\cup}$ and modular system operation

$\cup$: the former is a primitive module in modular system that performs the operation of union on its input vocabulary and generates some output while the latter is not a primitive module but is an operation that combines two modular systems.

Model-theoretic assumptions on MCS

For simplicity of the exposition, we focus on multi-context systems for which the belief sets BS_{L_i} of the logic L_i of each context $C_i = (L_i, kb_i, br_i)$ is a class of (relational) structures over some fixed vocabulary τ_i, where each structure corresponds to a belief set[2]. The vocabulary τ_i contains, in particular, all symbols that are associated with context C_i in some bridge rule of the multi-context system. Without loss of generality, we assume that $\tau_i \cap \tau_j = \emptyset$ for $i \neq j$. For a multi-context system $\mathit{MCS} = (C_1, \ldots, C_n)$, its vocabulary is the union of the vocabularies of its contexts, $\tau := \bigcup_{i=1}^{n} \tau_i$.

Translation T (from MCS to modular systems)

Let $\mathit{MCS} := (C_1, \ldots, C_n)$. For each vocabulary symbol P of *MCS*, introduce a new symbol P' of the same type and arity. We use these new symbols to create loops and simulate information propagation through bridge rules. Also, let us denote pairwise equalities on primed and unprimed symbols in C_i by $\overline{P'_i} = \overline{P_i}$. Then,

$$T[\mathit{MCS}] := (T^C[C_1] \cap \cdots \cap T^C[C_n])[\overline{P'_1} = \overline{P_1}] \ldots [\overline{P'_n} = \overline{P_n}],$$

Translation T^C of contexts

Translation of $C_i := \langle L_i, kb_i, br_i \rangle$ passes information from the translation of bridge rules to the translation of contexts:

$$T^C[C_i] := T^{br}[br_i] \rhd T^L[kb_i; C_i].$$

Translation T^L of knowledge bases

Let $C_i := \langle L_i, kb_i, br_i \rangle$ and let τ_i be the vocabulary of C_i. Also, let $H_1, \cdots, H_m$ be new symbols that represent heads of rules in br_i. Now, $T^L[kb_i; C_i]$ is a primitive module M with $\sigma_M = \{H_1, \cdots, H_m\}$, $\varepsilon_M = \tau_i$ and:

$$\mathcal{B} \in M \iff \mathcal{B}|_{\tau_i} \in ACC_L(kb \cup \{H_i(\bar{a}) \mid \bar{a} \in H_i^{\mathcal{B}}\}).$$

[2]Non-structural belief sets can be encoded as structures for the purpose of this translation but we are not concerned with this encoding here.

Translation T^{br} of bridge rules

Bridge rules are translated to model their computation using relational algebra, i.e., (1) complement the interpretation of negated literals using M_{Compl}, (2) join the interpretation of positive literals with the complemented interpretation of negative literals using $M_{\bowtie}$, (3) project the joint interpretation according to variables present in the head using M_{π}, and, (4) finally, use $M_{\cup}$ to compute the union of all projected interpretations for rules with the same symbol in their head. This is done by first partitioning br to subsets $br_1, \ldots, br_m$ based on the head of rules, i.e., $r, r' \in br_i \Leftrightarrow hd(r) = hd(r')$. Then, we have:

$$T^{br}[br] := T^{p}[br_1] \cap \cdots \cap T^{p}[br_m],$$
$$T^{p}[\{r_1, \ldots, r_k\}] := (T^{r}[r_1] \cap \cdots \cap T^{r}[r_k]) \rhd M^{p}_{\cup},$$

where $T^r[r]$ translates one bridge rule and $M^p_{\cup}$ performs the union operation[3]. The output vocabulary of module $M^p_{\cup}$ is the symbol H that appeared before. The input vocabulary of $M^p_{\cup}$ is $\{P_{r_1}, \ldots, P_{r_k}\}$ with each P_{r_i} denoting output of one rule construction.

Translation of rule r, $T^r[r]$, is obtained by composing $T^{body}[r]$ (translation of r's body) and M^r_{π} as follows:

$$T^{r}[r] := T^{body}[r] \rhd M^{r}_{\pi},$$
$$T^{body}[r] := (\textstyle\bigcap_{P \in body^{-}(r)} M^{P}_{Compl}) \rhd M^{r}_{\bowtie},$$

Translation of r's body, as before, is computed by joining either the predicates themselves or their complement (through M^P_{Compl}).

5.4.2 Support Functions of Primitive Modules in Translation T

Translation T that we gave in the previous section defined the structure of a modular system. However, in order to be able to use supported model semantics for this modular system, we should define a support function for every primitive module used in this translation. The primitive modules we used in translation T were divided into two categories: the primitive modules that performed relational algebraic operations, and the primitive modules that represented a context.

Here, we specify the support functions of all relational algebraic primitive modules and leave the specification of support functions for context modules to the next sections:

Complement: $Sup_{\neg}(\mathcal{B}) = \{e\}$ where e is the empty justification function as in Definition 5.9.

[3] $M^p_{\cup}$ performs a standard relational algebraic operation and should not be confused with the operation that combines modules.

Join: $Sup_{\bowtie}(\mathcal{B}) = \{j\}$ where $j(R_t)$ is the set of atoms $S_{t'}$ with $S \in \sigma_{M_{\bowtie}}$, $t' \in S^{\mathcal{B}}$, and $S(t') \in body^{+}(r)$ where r is the bridge rule represented by $M_{\bowtie}$.

Projection: Let $\sigma = \{S\}$ and $\varepsilon = \{R\}$ be the input and output vocabulary of the projection module. Then, $Sup_{\pi}(\mathcal{B})$ is the set of all mappings that take R_t to a set $\{S_{t'}\}$ where $t' \in S^{\mathcal{B}}$ and t' matches t on all projected columns (according to M_{π}).

Union: If $\sigma_{M_{\cup}} = \{S^1, \cdots, S^n\}$ and $\varepsilon_{M_{\cup}} = \{R\}$, then $Sup_{\cup}(\mathcal{B})$ is the set of all different mappings that take R_t to the singleton set $\{S^i_{t'}\}$ with $1 \leq i \leq n$ and $t \in S^{i^{\mathcal{B}}}$.

Intuitively, each justification function for an algebraic relational operation says why each tuple is present in the result of that operation according to the operator's semantics. For example, the semantics of algebraic operation π_1 is to keep just the first element of each tuple and discard the rest. Therefore, justification $j_{\pi_1}(R_a)$ can be either $\{S_{\langle a,b\rangle}\}$ or $\{S_{\langle a,b\rangle}\}$ but not $\{S_{\langle b,a\rangle}\}$.

5.4.3 Expressing Equilibrium Semantics of MCSs using Supported Model Semantics

In this part, we use the empty support function for supporting primitive modules that represented contexts in the translation given by T before. Now, we want to prove that the supported models of modular system obtained by $T[MCS]$ correctly represents *MCS* (under equilibria semantics). We use $vocab(S_i)$ to refer to the vocabulary of structure S_i.

Definition 5.10 *Consider belief state $S := (S_1, \cdots, S_n)$ such that $vocab(S_i) \cap vocab(S_j) = \emptyset$ (for $i \neq j$). Also, let $D_1, \cdots, D_n$ be n new unary predicate symbols. Structure $\mathcal{B}$ over vocabulary $\{D_1, \cdots, D_n\} \cup \bigcup_{i \in \{1,\cdots,n\}} vocab(S_i)$* represents belief state *S if: (1) $dom(\mathcal{B}) = dom(S_1) \cup \cdots \cup dom(S_n)$, (2) $D_i^{\mathcal{B}} = dom(S_i)$ for $i \in \{1, \cdots, n\}$, and, (3) $R^{\mathcal{B}} = R^{S_i}$ if $R \in vocab(S_i)$.*

Definition 5.11 *A modular system M* correctly represents multi-context system *$MCS = (C_1, \cdots, C_n)$* under equilibria semantics *if for all belief states $S = (S_1, \ldots, S_n)$ and its corresponding structure $\mathcal{B}$, $\mathcal{B} \in M$ iff S is an equilibrium of MCS.*

Theorem 5.2 *Let MCS be a multi-context system and $M := T[MCS]$. Also, let the relational algebraic primitive modules of M use support functions as in Section 5.4.2 and contextual primitive modules of M use the empty support function. Then, M correctly represents MCS under equilibrium semantics.*

Proof: ($\subseteq$) Let $S = (S_1, \cdots, S_n)$ be an equilibrium of multi-context system *MCS*. We prove that the structure $\mathcal{B}$ which represents S is a supported model of M. Firstly, a simple induction shows that

M has the empty support function for all its model. So, we only have to show that $\mathcal{B}$ is a model of M. By definition of representation, $\mathcal{B}|_{vocab(S_i)} = S_i$. Let $\mathcal{B}'$ be a structure such that $\mathcal{B}'|_{vocab(\mathcal{B})} = \mathcal{B}$ and $\mathcal{B}'$ satisfies the join, union, projection and complement primitive modules in $T[\mathit{MCS}]$. Note that such $\mathcal{B}'$ always uniquely exists (because join, union, projection and complement modules are total and deterministic). Now, note that the set of true atoms in $\mathcal{B}'|_{\sigma_{M_i}}$ is $\{head'(r) \mid r \in br_i$ and r applicable under $S\}$. Therefore, as S is an equilibrium of *MCS* and $vocab(S_i) = \sigma_{M_i} \cup \varepsilon_{M_i}$, we have that $\mathcal{B}'|_{\varepsilon_{M_i}} \in ACC_i(kb_i \cup \sigma^{\mathcal{B}}_{M_i}\}$. So, $\mathcal{B}'|_{vocab(M_i)} \in M_i$. Hence, $\mathcal{B} \in M$.
($\supseteq$) Let $\mathcal{B}$ be a supported model of M. By definition, $\mathcal{B}$ is also a model of M. We prove that the belief state $S = (S_1, \ldots, S_n)$ which is represented by $\mathcal{B}$ is an equilibrium of *MCS*. Because of join, union, projection and negation modules being total and deterministic, there exists unique $\mathcal{B}'$ which satisfies all the primitive modules of M and $\mathcal{B}'|_{vocab(\mathcal{B})} = \mathcal{B}$. Also, by semantics of these relational algebraic modules, we know that $at(\mathcal{B}'|_{\sigma_{M_i}}) = \{head'(r) \mid r \in br_i$ and r is applicable under $S\}$. Also, as $\mathcal{B}'|_{vocab(M_i)} \in M_i$, we know that $S_i \in ACC_i(kb_i \cup \sigma^{\mathcal{B}}_{M_i})$. Therefore, S is an equilibrium of *MCS*. ∎

Generalizing to rules with variables

The original definition of multi-context systems disallows bridge rules with variables, but when limited to contexts with structural belief sets (as in our case), the original definition can easily be generalized to accommodate variables under the usual assumptions of multi-sorted logics (that are needed because belief sets might have different domains). Such an extension of multi-context system is proposed in [71]. Theorem 5.2 is still correct under such extensions (i.e., if variables are present and usual assumptions on multi-sorted logics are guaranteed). This further proves the robustness of supported model semantics for modular systems.

5.4.4 Translating Grounded Equilibria

Another semantics for multi-context systems is the grounded equilibrium semantics that is defined for the *reducible* subset of multi-context systems. Here, we show that grounded equilibria can also be represented in the extended modular system semantics. Moreover, we use exactly the same translation T as in Section 5.4.1 except that, now, we use different support functions for the primitive modules that represent contexts in the translation T. Note that, here, we also use the same support functions for other relational algebraic primitive modules as given in Section 5.4.2.

So, it only remains to define support function of context modules. Since contexts are reducible

here, we use the reducibility condition to define the proper support function. By definition of T^L, $\sigma_M := \{H^1, \cdots, H^n\}$ (symbols occurring in the head of rules in br) and $\varepsilon_M := \{R^1, \cdots, R^m\}$ (symbols from context C that occur in the body of some bridge rule in the multi-context system). Now, define $Sup_M(\mathcal{B})$ to be the set of all mappings from R^i_a to a minimal set H' satisfying the following conditions:

$$H' \subseteq {H^1}^{\mathcal{B}} \cup \cdots \cup {H^n}^{\mathcal{B}} \text{ and,}$$
$$\{S\} = ACC(red_L(kb \cup H', \mathcal{B})) \Rightarrow \textstyle\bigwedge_{i\in\{1,\cdots,m\}} {R^i}^{\mathcal{B}} \subseteq S.$$

Now, we want to show that, using these support functions for primitive modules, supported models of $T[$*MCS*$]$ uniquely correspond to grounded equilibria of *MCS*. We first prove this property for the case of *MCS* being a definite multi-context system and then extend it to the general case of all reducible multi-context systems.

Proposition 5.2 *Let MCS be a definite multi-context system and $M := T[$MCS$]$. Also, let relational algebraic primitive modules of M be supported by functions given in Section 5.4.2 and context modules of M be supported by the minimality-based function above. Then, M has a unique supported model $\mathcal{B}$ that represents the unique minimal equilibrium of MCS.*

Proof: By Theorem 5.2, we know that the all equilibria of *MCS* are models of M. So, we only have to show that: (1) the unique minimal equilibrium of *MCS* is supported, and (2) M has only one supported model.

To prove the former, by [25], we know that $S := (S_1, \ldots, S_n)$ (where $\{S_i\} = ACC_i(kb_i^\infty)$) is the unique minimal equilibrium of *MCS*. Let $\mathcal{B}$ represent S, and let ordering $<_j$ be such that, for $R_t \in S_j$ and $R'_{t'} \in S_k$, we have $R_t <_j R'_{t'}$ if and only if ordinal α exists such that $R_t \in E^\alpha_j\}$ but $R'_{t'} \notin E^\alpha_k$ (where $\{E^\alpha_m\} = ACC(kb^r_m)$ as in [25]). Applied to Definition 5.7, ordering $<_j$ witnesses that $Sup_M(\mathcal{B}) \neq \emptyset$ and, thus, $\mathcal{B}$ is supported.

To prove the latter, let $\mathcal{B}$ be a supported model of M. Since $\mathcal{B}$ is a supported model, it is justified by a well-founded and non-circular justification function j. Let us define series $\mathcal{B}^\alpha$ as follows:

$$\mathcal{B}^0 := \{R_t \mid j(R_t) = \emptyset\},$$
$$\mathcal{B}^{\alpha+1} := \mathcal{B}^\alpha \cup \{R_t \mid j(R_t) \subseteq B^\alpha\},$$
$$\text{(if } \alpha \text{ is a limit ordinal) } \mathcal{B}^\alpha := \textstyle\bigcup_{\beta<\alpha} \mathcal{B}^\beta.$$

Since j is well-founded and non-circular, $at(\mathcal{B}) = \bigcup_\alpha \mathcal{B}^\alpha$. Now, we use a straightforward induction to show that, for all α, we have $\mathcal{B}^\alpha$ represents belief state $E^\alpha := (E^\alpha_1, \cdots, E^\alpha_n)$ with $\{E^\alpha_i\} =$

$ACC(kb_i^\alpha)$ as in [25]. Hence, $\mathcal{B}$ represents a minimal equilibrium of *MCS* and, since *MCS* has only one minimal equilibrium, only one such $\mathcal{B}$ exists. That is, M has only one supported model. ∎

Definition 5.12 *A supported modular system* M correctly represents a multi-context system $MCS = (C_1, C_2, \cdots, C_n)$ under grounded equilibrium semantics *if for all belief states* $S = (S_1, \cdots, S_n)$ *and its corresponding structure* $\mathcal{B}$*, we have* $\mathcal{B}$ *is a sup. model of* $M \Leftrightarrow S$ *is a grounded equilibrium of MCS.*

Theorem 5.3 *Let MCS be a reducible multi-context system and* $M := T[MCS]$ *be such that relational algebraic primitive modules of* M *are supported by functions given in Section 5.4.2 and context modules of* M *are supported by minimality-based support functions defined in this section. Then,* M *correctly represents MCS under grounded equilibrium semantics.*

Proof: Let M' be the positive part of M, i.e., the part without primitive modules for negation. Now, take grounded equilibrium S and its representation $\mathcal{B}$. By Proposition 5.2, $\mathcal{B}$ is the unique supported model of M'. Therefore, $\mathcal{B}$ is a supported model of M because all the outputs of the negation modules are always returns trivially supported (by definition of support function in negation modules).
Also, if $\mathcal{B}$ is a supported model of M then $\mathcal{B}$ is also a supported model of M' (defined as above). So, by Proposition 5.2, belief state S represented by $\mathcal{B}$ represents the unique minimal equilibrium of MCS^S and, therefore, a grounded equilibrium. ∎

Together, Theorems 5.2 and 5.3 show that both MCSs under equilibrium semantics and MCSs under grounded equilibrium semantics can be naturally translates into a modular system under supported model semantics. Thus, we showed that supported model semantics generalizes and unified the two different semantics of multi-context systems: its equilibrium semantics and its grounded equilibrium semantics.

5.5 Conclusion and Future Directions

In this chapter, we showed that the concept of support in modular systems is useful for naturally representing many sets of intended models. We also showed that modular system operators that combine more primitive modules can be easily and naturally extended to allow justification functions and support functions. We also showed that our new supported model semantics for modular systems generalizes our previous model-theoretical semantics for modular systems. In this chapter, we defined supported model semantics using a model-theoretical viewpoint. However, similar

to Chapter 4, we believe that a there exists a useful functional viewpoint that corresponds to out model-theoretical treatment of supported model semantics. Studying this functional viewpoint is a future direction to extend this research.

Moreover, in this chapter, we showed that supported model semantics for modular systems is expressive enough to unify two different semantics that have previously been defined for multi-context systems. Therefore, we now have a formal comparison between the expressive power of these modular systems and the expressive power of multi-context systems. Secondly, we showed that capturing the two different semantics of multi-context systems is doable by just changing the support function for the modules that represent a context. In this chapter, we studied only two of these support functions for context modules. It is left to study what other interesting and meaningful support functions can be used for the underlying primitive modules to obtain other useful semantics for multi-context systems.

More importantly, through relating modular systems to multi-context systems, we have provided the necessary means to cross-fertilize these two different but related frameworks for interlinking knowledge bases. For example, a consequence of Theorem 5.2 is that the search for equlibria in multi-context systems can be performed using the algorithm we developed in chapter 4 for finding solutions to modular systems. An interesting future research direction is to extend that algorithm to handle modular systems with supported vocabularies, and to apply it for finding grounded equilibria of multi-context systems.

Finally, another future direction that uses the results of this chapter is to develop diagnosis procedures similar to [23, 61] for modular systems under supported model semantics and thus helping users of modular system framework to find and fix bugs in a complicated modular system.

Chapter 6

Supported Semantics for Propositional Logic Programs

6.1 Introduction

Answer set programs constitute one of the most important declarative programming paradigms that are readily available and actively being expanded. Stable model semantics is the semantics behind the success of answer set programming. Stable model semantics is proven to be suitable for declarative specification of many tasks, specifically those that require non-monotonic reasoning or reasoning about beliefs of agents. Due to this success, there has been many efforts to extend this semantics to a more general class of programs (i.e., to lift the restriction on answer set programs to have the form of either a normal or a disjunctive logic program). Such efforts mainly fall into two categories: (1) those with a practical goal to extend the language of answer set programs with specific constructs such as aggregates, and, (2) those with a fundamental approach that bring new insights into the stable models themselves.

This chapter falls into the second category above because we are not motivated by a particular construct for which a semantics is needed. Instead, what motivates us is a combination of the following: (1) pure interest in the philosophy of stable models, (2) practical need to extend stable models to the full propositional language, and, (3) the current shortcomings of existing extensions of stable model semantics.

We believe that the main philosophical principle behind the development of stable models (and the reason for its success) is the *principle of rationality*: all atoms in all stable models are justified

and justifications are well-founded (not circular). This principle is sometimes known as the property of being strongly grounded and is, of course, useful when modeling the set of possible belief sets of rational agents. It was first noted in [154] that one of the places where non-monotonic reasoning excels is to model the process of reasoning about knowledge and, thus, autoepistemic reasoning was introduced. Later, in their seminal paper introducing stable model semantics, Gelfond and Lifschitz [89] noted that stable models also serve the same purpose. Therefore, we expect extensions of stable model semantics to also adhere to the same philosophical principle and disallow self-justified models.

As we see in Examples 6.1 and 6.2, neither of the two main extensions of stable model semantics (i.e., equilibrium model semantics [164] and FLP semantics [67]) guarantee this desired property. Our examples demonstrate cases where FLP semantics and equilibrium models do not agree. We argue that, in the first case, the natural interpretation is the one given by FLP semantics, while, in the second case, equilibrium models give the more natural interpretation. The main achievement of this chapter is to define a new extension of stable model semantics that works flawlessly in all cases.

Example 6.1 *Consider program* Π_1 *as follows:*

$$\Pi_1 := \left\{ \begin{array}{l} a \leftarrow \textbf{not not } b. \\ b \leftarrow a. \end{array} \right\}. \tag{6.1}$$

Π_1 *is neither a normal nor a disjunctive logic program (because of "***not not** b*" in the body of its first rule). Thus, stable model semantics is not applicable to* Π_1*. However, both equilibrium model semantics and FLP semantics are applicable to* Π_1*. Both these semantics agree that* Π_1 *has an intended model* $S_1 := \emptyset$ *but, according to equilibrium models,* Π_1 *has yet another intended model* $S_2 := \{a, b\}$*. We want to argue that* S_2 *suffers from self-justification and, hence, is not rational.*

Equilibrium model S_2 *asserts that both* a *and* b *are believed so they should both be justified. However, according to* Π_1*, the only possible justification for* a *is "***not not** b*". Thus,* **not not** b *should be believed and its justification should not depend on* a *(otherwise, it would constitute a self-justification). To believe in "***not not** b*",* **not** b *should not be believable, i.e., every way to believe in* b *should fail. Now, since the only way to believe* b *is to first believe in* a *and since this only way constitutes a self-justification,* b *cannot be believed. Thus, we cannot believe in "***not not** b*" and, hence, we cannot believe in* a *either. The following self-justified chain summarizes our discussion:*

$$a \Rightarrow (\textbf{not not } b) \Rightarrow b \Rightarrow a$$

Above argument shows that S_2 *is self-justified (also sometimes known as circularly justified) and it should not be an intended model of* Π_1*. Thus, a a faithful extension of stable model semantics can*

only allow S_1. So, in this example, FLP semantics works as desired and captures the right model. Equilibrium model semantics, on the other hand, allows non-intended models.

*Before moving on, consider normal logic program $\Pi_1' := \{a \leftarrow$ **not** $c.\ ,\ \ b \leftarrow a.\ ,\ \ c \leftarrow$ **not** $b.\}$. Stable models of Π_1' are $S_1' := \{c\}$ and $S_2' := \{a, b\}$. We like to point out that, in Π_1', model S_2' is justified because it assumes "***not*** c" and so it can deduce a and b. In terms of program Π_1, however, this is like assuming the existence of some secret way to justify* **not not** b. *Such an assumption contradicts the closed-world assumption. Hence, Π_1 differs from Π_1' in that Π_1' adds a new way to justify* **not not** b *that does not exist in Π_1.*

Note that there are other semantics that extend stable models (such as Ferraris [70]) that agree with equilibrium models over the class of propositional programs. Clearly, such semantics also allow self-justification in program Π_1 of Example 6.1. Our next example discusses a case where FLP semantics allows self-justification while equilibrium models disallow it:

Example 6.2 *Consider program Π_2 as follows:*

$$\Pi_2 := \left\{ \begin{array}{l} a \leftarrow b. \\ b \leftarrow (a \vee \textbf{not}\ a). \end{array} \right\}. \tag{6.2}$$

Since program Π_2 is not in the recognized syntax of the original definition of FLP semantics, in this example, we consider an extension of FLP semantics defined in [197]. In Section 6.4, we consider yet another (but different) extension of FLP semantics.

According to [197], Π_2 has a single intended model $T := \{a, b\}$. However, according to Π_2, the only possible justification for atom a is atom b. Also, atom b can only be justified by formula "$a \vee$ **not** a*". However, since "*__not__ a*" is not true in model T, the only possible justification for "$a \vee$* **not** a*" is a itself. Therefore, model T suffers from the following self-justified chain:*

$$a \Rightarrow b \Rightarrow (a \vee \textbf{not}\ a) \Rightarrow a$$

Hence, we believe that T cannot be an intended model of Π_2 and that Π_2 should not have any intended models. This view agrees with the interpretation given by equilibrium models for Π_2. Thus, in this example, equilibrium model semantics captures the right intended models while FLP semantics allows self-justification.

Examples 6.1 and 6.2 show that neither equilibrium model semantics nor FLP semantics do not capture the right intended models everywhere. This chapter extends stable model semantics in a

way that is faithful to its founding principle of only allowing rational belief sets. Our semantics has the following properties:

(1) It is *defined for full propositional language*,

(2) It *extends stable model semantics*, and,

(3) It *represents the possible belief sets of a rational agent*, i.e., all beliefs are justified and self-justifications are disallowed.

The semantics we propose is based on intuitionistic derivability and, thus, it *enables us to use the full force of intuitionistic logic to study stable models*.

In what follows, Section 6.2 reviews the necessary background. Section 6.3 introduces supported model semantics and gives a proof that supported model semantics indeed extends stable model semantics (i.e., on the class of normal/disjunctive logic programs, they coincide). Section 6.3 also gives an important characterization of supported models in terms of Kripke structures. Section 6.4 relates our semantics to other non-monotonic semantics. We prove that all supported models are minimal classical models, Clark completion models, and, most importantly, equilibrium models (note that the other direction does not hold in general). Finally, Section 6.5 characterizes the complexity of different reasoning tasks in supported model semantics. Our results show that such tasks remain as computationally complex as similar tasks in other extensions of stable model semantics.

6.2 Background

Here, we review the required background of this chapter.

Logic programs: are sets of rules r of form:

$$h_1; \cdots ; h_m \leftarrow p_1, \cdots , p_i, \textbf{not } p_{i+1}, \cdots , \textbf{not } p_n. \tag{6.3}$$

where $0 \leq i \leq n$. $h_1, \cdots , h_m$ are propositional atoms called *heads* of r and $p_1, \cdots , p_n$ are propositional atoms forming the *body* of r. A rule r is (a) *normal* if $m = 1$, (b) *disjunctive* if $m > 1$, and, (c) a *constraint* if $m = 0$. Normal logic programs (NLP) can only have normal rules but disjunctive logic programs (DLP) can have both normal and disjunctive rules. A constraint can be expressed using normal rules and, thus, can be included in both NLPs and DLPs. The intuitive reading of Rule (6.3) is that if all atoms $p_1, \cdots , p_i$ are believed and none of the atoms $p_{i+1}, \cdots , p_n$ can be believed, then there is a reason to believe at least one of the atoms $h_1, \cdots , h_m$.

Propositional programs: are sets of arbitrary propositional formulas. A propositional formula is a formula constructed using propositional atoms $a, b, c, \cdots$, binary operators $\wedge$, $\vee$ and $\rightarrow$, and

zero-ary constant $\bot$. The propositional formula representing Rule (6.3) is:

$$p_1 \wedge ... \wedge p_i \wedge (p_{i+1} \rightarrow \bot) \wedge ... \wedge (p_n \rightarrow \bot) \rightarrow h_1 \vee ... \vee h_m. \tag{6.4}$$

We also use operators $\leftarrow$, $\neg$ and **not** as syntactical variants of $\rightarrow$, i.e., $(\phi \leftarrow \psi) := (\psi \rightarrow \phi)$ and $(\textbf{not}\ \phi) := \neg\phi := \phi \rightarrow \bot$. Moreover, we define $\top := (\bot \rightarrow \bot)$ and $\neg.A := \{\neg a \mid a \in A\}$ (A is a set of propositional atoms). Furthermore, for propositional program Π, $vocab(\Pi)$ denotes the set of all propositional atoms used in Π.

Example 6.3 *Propositional programs Π_1 and Π_2 from Examples 6.1 and 6.2 can be represented as follows:*

$$\begin{array}{c} \Pi_1 := \{((b \rightarrow \bot) \rightarrow \bot) \rightarrow a.\ ,\ \ a \rightarrow b.\}, \\ \Pi_2 := \{b \rightarrow a.\ ,\ \ (a \vee (a \rightarrow \bot)) \rightarrow b.\}. \end{array} \tag{6.5}$$

Here, $vocab(\Pi_1) = vocab(\Pi_2) = \{a, b\}$.

Stable model semantics: is a semantics for logic programs. Let Π be a DLP and S a set of propositional atoms. Π^S is the set of *positive rules* "$h_1; \cdots ; h_m \leftarrow p_1, \cdots , p_i$" s.t. a rule "$h_1; \cdots ; h_m \leftarrow p_1, \cdots , p_i, \textbf{not}\ p_{i+1}, \cdots , \textbf{not}\ p_n$" exists in Π with $S \cap \{p_{i+1}, \cdots , p_n\} = \emptyset$, i.e., Π^S is the positive part of Π that remains applicable according to S. S is a *stable model* of Π iff S is a minimal classical model of Π^S.

Equilibrium model semantics: extends stable models to full propositional language and is defined using satisfiability in logic of *here and there (HT-logic)*. In HT-logic, HT-model is a pair $\langle H, T\rangle$ where $H \subseteq T \subseteq U$ (U: the universe of propositional atoms). In HT-logic, $\langle H, T\rangle \models_{\text{HT}} \phi$ if: 1. ϕ is a propositional atom a and $a \in H$, 2. $\phi := (\phi_1 \wedge \phi_2)$ and $\langle H, T\rangle \models_{\text{HT}} \phi_1$ and $\langle H, T\rangle \models_{\text{HT}} \phi_2$, 3. $\phi := (\phi_1 \vee \phi_2)$ and either $\langle H, T\rangle \models_{\text{HT}} \phi_1$ or $\langle H, T\rangle \models_{\text{HT}} \phi_2$, or, 4. $\phi := (\phi_1 \rightarrow \phi_2)$ and both of the following hold: (a) if $\langle H, T\rangle \models_{\text{HT}} \phi_1$ then $\langle H, T\rangle \models_{\text{HT}} \phi_2$, and, (b) if $\langle T, T\rangle \models_{\text{HT}} \phi_1$ then $\langle T, T\rangle \models_{\text{HT}} \phi_2$. Also, for a propositional program Π, $\langle H, T\rangle \models_{\text{HT}} \Pi$ if $\langle H, T\rangle \models_{\text{HT}} \phi$ for all $\phi \in P$. Finally, a set S of propositional atoms is an *equilibrium model* of Π if (1) $\langle S, S\rangle \models_{\text{HT}} \Pi$, and, (2) for all $S' \subsetneq S$: $\langle S', S\rangle \not\models_{\text{HT}} \Pi$.

Interested reader is referred to [13] for a thorough review of stable models and to [164] (resp. to [67]) for a detailed discussion on equilibrium models (resp. FLP semantics). Also, [134] compactly (and usefully) reviews thirteen different ways of defining stable models.

Intuitionistic logic: is a subset of classical logic without the law of excluded middle. Here, $Con_I(\Gamma)$ (resp. $Con(\Gamma)$) denotes intuitionistic (resp. classical) consequences of Γ.

6.3 Supported Models

This section introduces the notion of supported models in terms of a form of completion in intuitionistic logic. We start by a simple definition of intended models (that we call supported models) and, thanks to the vast literature on intuitionistic logic, we are able to characterize supported models in terms of Kripke models. The latter characterization is extremely useful when we relate supported models to other non-monotonic semantics.

Definition 6.1 (Supported Models) *Let Π be a propositional program and let U be the universe of propositions (particularly, $vocab(\Pi) \subseteq U$). Then, for $S \subseteq U$, we say that S* is a supported model of Π w.r.t. universe U *iff (1) $\bot \notin Con_I(\Pi \cup \neg.(U \setminus S))$, and, (2) $S \subseteq Con_I(\Pi \cup \neg.(U \setminus S))$.*

The intuition behind Definition 6.1 is that a model S is supported in a program Π if the justification for inclusion of all propositions asserted by S can be traced back to one of the following primitive justifications (without circularity):

1. Something that is intuitionistically trivial,
2. Something that is asserted by Π, or,
3. Nonbelief in a propositional atom.

The inclusion of the last primitive justification above is motivated by the closed-world assumption and the innate justification it brings for nonbelief in a propositional atom. Moreover, it is also a widely accepted philosophical principle that the burden of proof can only be expected for propositions that are asserted, i.e., propositional atoms in S. Thus, Definition 6.1 says that a model is supported if, firstly, nonbelief in excluded atoms does not make us inconsistent and, secondly, included atoms are justified (without circularity) by the program Π and nonbelief in excluded atoms. Non-circularity of justifications in Definition 6.1 is guaranteed by the well-founded-ness and finiteness of their intuitionistic proofs.

Example 6.4 *Consider program Π_1 of Example 6.1. Firstly, $S_1 = \emptyset$ is supported because $\Pi_1 \cup \{\neg a, \neg b\}$ is consistent. Secondly, $S_2 = \{a, b\}$ is not supported because $a \notin Con_I(\Pi_1)$.*

Similarly, consider program Π_2 of Example 6.2. Model $T = \{a, b\}$ is not supported because $a, b \notin Con_I(\Pi_2)$ (note that a and b are classically derivable from Π_2). Intuitionistic underivability of a and b is shown using HT-model $\langle \emptyset, \{a, b\} \rangle$ that satisfies Π_2 but not a or b.

We now want to prove that supported models extend stable models, but we first need a lemma and a definition:

Lemma 6.1 *For positive DLP* Π *(i.e.,* Π *has no negative literal in the body of its rules) and universe* U *of propositional atoms, we have that for all* $S \subseteq U$*,* S *is supported in* Π *iff* S *is a minimal classical model of* Π*.*

Proof: Define $\Pi' := \Pi \cup \neg.(U \setminus S)$.
($\Rightarrow$) By S being a supported model of Π, we know $S \subseteq Con_I(\Pi') \subseteq Con(\Pi')$. Also, since $\bot \in Con_I(\Gamma)$ iff $\bot \in Con(\Gamma)$ (for any propositional theory Γ [121]), we know that Π is classically satisfiable. Therefore, S is a minimal classical model of Π.
($\Leftarrow$) Let S be a minimal classical model of Π. Since equilibrium models agree with stable models on DLPs, S is an equilibrium model of Π too. Now, assume that S is not supported. Then, there exists a Kripke model $\langle \mathcal{K}, \alpha \rangle$ s.t. $\alpha \Vdash \Pi'$ but $\alpha \not\Vdash S$. Since S is a minimal classical model of Π' and since every $\beta \geq \alpha$ forces Π', we know that, all maximal states $\beta \geq \alpha$ force S. Now, take maximal state $\gamma \geq \alpha$ s.t. $\gamma \not\Vdash S$ (i.e., if $\gamma' > \gamma$ then $\gamma' \Vdash S$). Define $H := \{a \in U \mid \gamma \Vdash a\}$. Since $\langle H, S \rangle \models_{\text{HT}} \Pi'$ and $H \subsetneq S$, our assumption of S being an equilibrium model is contradicted. Hence, S is supported. ∎

Definition 6.2 (Classical Kripke Model) *Let* S *be a set of propositional atoms and let* $\langle \mathcal{K}, \alpha \rangle$ *with* $\mathcal{K} := \langle W, \leq, \Vdash \rangle$*,* $W := \{\alpha\}$*, and* $\leq := \emptyset$ *be a Kripke model such that* α *forces a propositional atom* a *if and only if* $a \in S$*. Then,* $\langle \mathcal{K}, \alpha \rangle$ *is known as the* Kripke representation of classical model S *(or, shortly, the* classical Kripke model S*).*

Note that, for all classical models S, $S \models \Pi$ iff the Kripke representation $\langle \mathcal{K}, \alpha \rangle$ of S has $\alpha \Vdash \Pi$.

Theorem 6.1 (Stable Model = Supported Model (in LP's)) *Let* Π *be an NLP/DLP and* U *be the universe of propositional atoms. Then, for* $S \subseteq U$*,* S *is a stable model of* Π *iff* S *is a supported model of* Π *w.r.t.* U*.*

Proof: Let $\Pi' := \Pi \cup \neg.(U \setminus S)$ and $\Pi'' := \Pi^S \cup \neg.(U \setminus S)$.
($\Rightarrow$) If S is a stable model then, firstly, S is a classical model and thus a classical Kripke model of Π. Therefore, Π' is consistent. Secondly, by Lemma 6.1, we know that S is a supported model of Π^S. Therefore, $S \subseteq Con_I(\Pi'') \subseteq Con_I(\Pi')$. Thus, S is supported.
($\Leftarrow$) Let S be a supported model of Π. Then, firstly, Π'' is consistent and thus Π' is also consistent. Secondly, we know that under assumptions $\neg.(U \setminus S)$, program Π is intuitionistically equivalent to program Π^S. Hence, $S \subseteq Con_I(\Pi'')$ and thus a supported model of Π^S. Now, by Lemma 6.1, S is a minimal model of Π^S and thus a stable model of Π. ∎

Since supported models are defined in terms of intuitionistic reasoning, the full force of intuitionistic logic can be used to study supported models. In particular, we want to characterize supported models using Kripke models of a propositional program.

Theorem 6.2 (Characterizing Supported Models) *For propositional program* Π *and finite universe* U *of propositional atoms, we have: for all* $S \subseteq U$*,* S *is a supported model of* Π *w.r.t.* U *iff (1) the Kripke representation of* S *forces* Π*, and, (2) for all Kripke models* $\langle \mathcal{K}, \alpha \rangle$ *of* $\Pi \cup \neg.(U \setminus S)$*, all* $\beta \geq \alpha$*, and all* $a \in S$*, we have* $\beta \Vdash a$*.*

Proof: ($\Rightarrow$) Let S be a supported model of Π. Then, $\Pi \cup \neg.(U \setminus S)$ is consistent. Therefore, S is a classical model of Π and, thus, the Kripke representation of S forces Π. Moreover, since $S \subseteq Con_I(\Pi \cup \neg.(U \setminus S))$, for all Kripke models $\langle \mathcal{K}, \alpha \rangle$ of $\Pi \cup \neg.(U \setminus S)$, $\alpha \Vdash a$. Thus, for all $\beta \geq \alpha$, and all $a \in S$, $\beta \Vdash a$.
($\Leftarrow$) Let S be a classical Kripke model of Π s.t. for all Kripke models $\langle \mathcal{K}, \alpha \rangle$ of $\Pi' := \Pi \cup \neg.(U \setminus S)$, all $\beta \geq \alpha$, and all $a \in S$, we have $\beta \Vdash a$. Then, S is a classical model of Π and, so, a classical model of Π'. Thus, firstly, Π' is consistent and, secondly, every $a \in S$ is true in all Kripke models of Π' and, hence, an intuitionistic consequence of Π'. Therefore, S is a supported model of Π. ∎

A nice consequence of Theorem 6.2 is the property that all supported models are minimal classical models:

Corollary 6.1 (Supported $\subseteq$ Minimal Classical) *Let* Π *be a propositional program with finite vocabulary and let* S *be a supported model of* Π*. Then,* S *is a minimal classical model of* Π*.*

Remember Example 6.1 from Section 6.1. Corollary 6.1 shows that, indeed, S_2 is not supported (since it is not a minimal model). Moreover, Corollary 6.1 connects supported models to minimal classical models. Next section connects supported models to other non-monotonic semantics in a similar way.

6.4 Relation to Existing Non-monotonic Semantics

This section connects our supported model semantics to other frameworks for non-monotonic reasoning. Perhaps, due to the wide acceptance of FLP semantics [67] and equilibrium model semantics [164], the most important question that must be answered here is how supported models relate to these two semantics.

We first start by the relation between our supported semantics and equilibrium models. As pointed out by [164], HT-logic is an intermediate logic (a logic that lies between classical and intuitionistic logic). Therefore, it is no surprise that a very natural connection exists between equilibrium models and supported models. Indeed, an HT-model is a special kind of Kripke models with only two states H and T such that $H \leq T$. According to [164], S is an equilibrium model of P if $\langle S, S\rangle \models_{\text{HT}} P$ but for all $S' \subsetneq S$: $\langle S', S\rangle \not\models_{\text{HT}} P$.

So, clearly, all supported models are equilibrium models too because (1) they are classical Kripke models, and, (2) they are the only minimal Kripke models. The following Theorem states exactly this fact:

Theorem 6.3 (Supported Models $\subseteq$ Equilibrium Models) *For propositional program Π with finite universe, every supported model is also an equilibrium model.*

Note that the inverse of Theorem 6.3 does not hold, i.e., there exist equilibrium models that are not supported. One such equilibrium model is, indeed, model S_2 from Example 6.1. Theorem 6.4 shows that HT-models (unlike equilibrium models) can completely characterize supportedness:

Theorem 6.4 *For propositional program Π over a finite universe U, we have that, for every $S \subseteq U$, S is supported iff $\langle S, S\rangle$ is the unique HT-model of $\Pi \cup \neg.(U \setminus S)$.*

Proof: ($\Rightarrow$) Let S be a supported model of Π. Then, by Corollary 6.1, S is a classical model of $\Pi' := \Pi \cup \neg.(U \setminus S)$ and thus $\langle S, S\rangle$ is an HT-model of Π'. Also, for all HT-models $\langle H, T\rangle$ of Π', by Theorem 6.2, we have $H = T = S$.
($\Leftarrow$) Let S be the unique HT-model of $\Pi' := \Pi \cup \neg.(U \setminus S)$. Assume that we have a Kripke model $\langle \mathcal{K}, \alpha\rangle$ of Π', a state $\beta \geq \alpha$ and some $a \in S$ such that $\beta \not\Vdash a$. Take maximal state $\gamma \geq \beta$ such that $\gamma \not\Vdash S$ (i.e., every state $\gamma' > \gamma$ forces S). Define $H := \{a \in U \mid \gamma \Vdash a\}$. Now, if γ is a maximal state in $\mathcal{K}$ (i.e., there is no state $\gamma' > \gamma$) then consider $M := \langle H, H\rangle$ and, otherwise, consider $M := \langle H, S\rangle$. Since $M \models_{\text{HT}} \Pi'$ and $H \subsetneq S$, it contradicts our assumption of $\langle S, S\rangle$ is the unique HT-model of Π'. Therefore, by Theorem 6.2, S has to be supported. ∎

Another interesting semantics for negation as failure in logic programs is given by Clark [36]. The Clark completion idea can be generalized to include rules with fully propositional bodies (instead of just a conjunction of literals). It so happens that supported models are also models of the Clark completion of a program, i.e., for a program Π consisting of only rules with an atom as their head, if an atom a is in a supported model S, then there should exist a rule $r \in P$ with $head(r) = a$ such that S classically satisfies $body(r)$. The following theorem states this fact formally:

Theorem 6.5 (Supported Models $\subseteq$ Clark Completion Models) *Let Π be a finite program and U a finite universe of propositional atoms. Then, all supported models of Π are also Clark completion models of Π.*

Note that Theorem 6.5 can also be viewed as a consequence of Theorem 6.3 because all equilibrium models of a program are also models of its Clark completion. Indeed, the proof of Theorem 6.5 shows that not being a model of Clark completion guarantees the existence of a minimal Kripke model with only two states (i.e., an HT-model $\langle S', S\rangle$ with $S' \subsetneq S$) that contradicts our characterization of supported models in terms of Kripke models.

Now, let us turn to FLP semantics. The intersection of the syntax of propositional programs and the syntax for which the original FLP semantics is exactly the syntax of disjunctive logic programs. On this limited intersection of syntax, FLP semantics coincides with stable model semantics and, thus, coincides with our supported semantics. However, there are extensions of FLP semantics that generalize the idea of FLP reducts to a more extensive syntax. Remember Example 6.2 that used an extension of FLP semantics due to [197]. In Example 6.2, the main reason why model T was considered an intended model was that "$a \vee$ **not** a" was assumed to be supported (something that cannot be assumed in intuitionistic logic for example). Indeed, in stable model semantics, one cannot assume such a thing. For example, consider program $\Pi := \{a \leftarrow \textbf{not}\ a\}$. According to stable model semantics, Π has no stable model and, thus, should not have any supported model either. However, assuming that $(a \vee \textbf{not}\ a)$ is supported makes us believe that model $S := \{a\}$ is supported. So, although classically valid, $(a \vee \textbf{not}\ a)$ is not supported.

The last semantics that we discuss in this section is yet another extension of FLP semantics. This extension is called the semantics of well-justified FLP answer sets [173]. This semantics is different from the extension given in [197] even over the class of propositional programs. The motivation behind development of [173] is very similar to ours but their method to address their concern is completely different.

Shen and You [174] noticed that standard FLP semantics suffers from self-justification. They addressed this deficiency using an approach similar to that of default logic (over a limited syntax of logic programs with C-atoms). Their approach was extended to other syntactical fragments in [171, 172]. Recently, their approach was extended to general logic programs [173] that also covers our intended syntactical fragment (full propositional programs).

Although the motivation behind [173] closely corresponds to ours (i.e., we both aim to remove circular justifications), the approach that we take towards this goal is completely different. We start from our definition that is given in terms of intuitionistic derivability and is completely declarative

(i.e., intended models are characterized in terms of their properties and not operationally). However, [173] takes an operational approach and uses an immediate consequence operator (a generalization of van Emden-Kowalski one-step provability operator [201]) to describe their intended models.

Apart from the methodological difference above, there also exist propositional programs that have different intended models according to our supported semantics relative to well-justified FLP semantics. For example, consider the following program Π_3:

$$\Pi_3 := \left\{ \begin{array}{l} a \leftarrow \neg a. \\ (a \vee \neg a) \leftarrow \top. \end{array} \right\}. \tag{6.6}$$

Firstly, note that program Π_3 above is acceptable in both our syntax and the syntax of general logic programs (as in [173]). Secondly, note that the second rule "$(a \vee \neg a) \leftarrow \top$", although trivial in classical logic, is not purposeless in program Π_3 above.

Program Π_3 in Equation (6.6) is intuitively read as follows: second rule says that either a or $\neg a$ has to be believed. However, according to the first rule, even if $\neg a$ is believed, a has to be believed too. So, since one of a or $\neg a$ has to be believed and since in both cases a is believed, according to Π_3, a must be believed. Thus, model $S := \{a\}$ is the only intended model of Π_3.

Indeed, in supported semantics, $\{a\}$ is the only supported model of Π_3. However, according to the well-justified FLP semantics, Π_3 has no well-justified model and here is why:

According to well-justified FLP semantics [173], $f\Pi_3^S = \{(a \vee \neg a) \leftarrow \top\}$ because the body of the rule "$a \leftarrow \neg a$" is not satisfied by S and thus this rule is removed. Since $S = \{a\}$, we have that $S^- = \emptyset$ and, thus, $\neg S^- = \emptyset$. Therefore, $T^{\alpha}_{f\Pi_3^S}(\emptyset, \neg S^-) = T^{\alpha}_{f\Pi_3^S}(\emptyset, \emptyset) = \{(a \vee \neg a)\}$[1]. Now, according to well-justified FLP semantics, S is accepted if and only if S is a classical consequence of $T^{\alpha}_{f\Pi_3^S}(\emptyset, \neg S^-) \cup \neg S^-$. However, since $\neg S^- = \emptyset$, we have that $T^{\alpha}_{f\Pi_3^S}(\emptyset, \neg S^-) \cup \neg S^- = \{(a \vee \neg a)\}$. Obviously, $a \in S$ is not a consequence of "$a \vee \neg a$" and, thus, S is not an intended model of Π_3 according to well-justified FLP semantics introduced in [173].

Discussions above show that our semantics is more suited to extend stable models to the full propositional language. A more thorough investigation of how our supported semantics relates to different extensions of FLP semantics is still needed.

[1] $T^{\alpha}_{f\Pi^I}(\emptyset, \neg I^-)$ is the least fixpoint of an operator that generalizes van Emeden-Kowalski one-step provability operator and is introduced in [173].

6.5 Complexity

We defined supported semantics and proved some natural connections between supported semantics and other well-established semantics including stable models, Kripke models, HT-models, equilibrium models, minimal classical models and Clark completion models. Except for Kripke models, deciding about the validity, satisfiability or falsity of a program in those other semantics are relatively easy (decidable in the second level of polynomial hierarchy in the worst case). However, for Kripke models, deciding whether a propositional program Π is a tautology (i.e., Π is true in all Kripke models) is PSPACE-complete [127, 180].

In this section we investigate the computational complexity of (1) checking whether a propositional model S is a supported model of a propositional program Π with respect to a universe U of propositional atoms, and, (2) doing cautious/brave reasoning for a propositional program P. Obviously, in this section, we assume that all three parts P, S and U are finite.

Let us first review what we already know about the computational complexity of finding a supported model (or checking if a given S is supported). On one hand, as disjunctive stable models coincide with supported models over the class of disjunctive logic programs, we know that (1) checking supported-ness of a given S is Π_1^P-hard, (2) brave reasoning about supported models is Σ_2^P-hard, and, (3) cautious reasoning about supported models is Π_2^P-hard.

On the other hand, by PSPACE-completeness of checking if a formula is an intuitionistic tautology, we also know that checking if a given S is a supported model can be done in PSPACE. The following theorem closes the gap between our lower bound (second level of PH) and upper bound (PSPACE) by proving that it is Π_1^P-complete to decide if a given S is supported:

Theorem 6.6 (Complexity of Decision Procedure) *Let P be a finite propositional program, U be a finite universe of propositional atoms and S be a subset of U. Then, the problem of deciding whether S is a supported model of P with respect to universe U is a Π_1^P-complete problem.*

Using Theorem 6.6, we can also characterize the complexity of brave/cautious reasoning about supported models:

Corollary 6.2 (Complexity of Brave/Cautious Reasoning) *Let P be a finite propositional program and U be a finite universe of propositional atoms and $S_T, S_F \subseteq U$. Then,*

1. *it is a Σ_2^P-complete task to decide if there exists S such that (1) $S_T \subseteq S$, (2) $S \cap S_F = \emptyset$, and (3) S is a supported model of P with respect to universe U (brave reasoning).*

2. *it is a Π_2^P-complete task to decide if all supported models S of P with respect to universe U satisfy both $S_T \subseteq S$ and $S \cap S_F = \emptyset$ (cautious reasoning).*

6.6 Conclusion and Future Works

Motivated by the non-circularity and well-foundedness of justifications for supported models of a modular system, in this chapter, we extended the well-established stable model semantics of logic programs to the full propositional logic while still disallowing self-justifications. Needless to mention that, while stable model semantics has been previously extended to full propositional logic, all the previous extensions from self-justification. However, the novelty of our approach is to start with the fundamental ideas that motivated stable models in the first place. Starting from such fundamentals, equivalence of supported semantics and stable model semantics (over the class of logic programs) is a necessary consequence of our approach rather than a surprising coincidence as in previous extensions.

This chapter also studied the connection between some of the proposed semantics for non-monotonic reasoning and our supported model semantics. There are still many semantics for non-monotonic reasoning (such as default logic and autoepistemic logic) for which the relation to our supported model semantics is not known. We hope that a detailed study of this relation reveals a deep and intrinsic connection between our semantics and other well-established frameworks for non-monotonic reasoning.

The last topic that we covered in this chapter is the complexity of reasoning about supported models. We proved that although we used intuitionistic reasoning to guarantee supportedness of models, none of the interesting reasoning tasks becomes more computationally complex than before (when supportedness was not a concern).

One of our future topics of interest is to develop an efficient mechanism to reason about supported models of a propositional formula. Another future direction is to study the exact relation between supported models of modular systems and the supported model semantics for propositional logic programs. We believe that there is an inherent connection between these two semantics and the results from one domain can be easily carried over to the other domain.

Chapter 7

Discussion and Future Directions

In this thesis, we touched on two fundamental issues of declarative programming in knowledge representation: (1) the problem of using arithmetic in declarative programming in a controlled manner, and, (2) the problem of modular declarative representation. We will discuss these two general directions and their possible extensions separately in the two following sections of this chapter.

7.1 Arithmetical Search Problems

In Chapter 3, we showed the shortcomings of existing declarative knowledge representation languages. We showed that many natural arithmetical problems cannot be expressed by existing declarative languages for knowledge representation despite the fact these problems are computationally within the bounds of such declarative KR languages. We also introduced a fragment of first-order logic that captures exactly the set of NP-recognizable arithmetical problems.

Benefits: A Summary

As we described in Chapter 3, the benefits of our study is manifold:

Guaranteeing Universality. Studying expressiveness of a knowledge representation problem guarantees the existence of specifications for certain classes of problems. For instance, our expressiveness results for knowledge representation languages in Chapter 3 showed that classes of small-cost arithmetical structures that are recognizable in NP are guaranteed to have representations in many of the existing declarative languages. Given the intuitiveness of the notion of being a small-cost structure, users can check if the problem they are specifying satisfies

small-cost condition and, if so, our expressiveness proofs can be used as guidelines on how a standard specification can be obtained.

Identifying Possible Extensions. Studying non-expressiveness of a knowledge representation language pinpoints the type of constructs that a language lacks. For example, our study of arithmetic in existing knowledge representation languages showed that these languages generally lack constructs for dealing with the number of digits of an input numbers. Thus, designers of knowledge representation languages can use our non-expressiveness results for equipping their knowledge representation languages with the right constructs that are guaranteed to increase the expressiveness of their language.

Solving Effectively. By studying characterization results with respect to a complexity class, one can identify cases when certain reasoning mechanisms with a less resource consumption footprint suffice for solving a problem. In these cases, such reasoning mechanisms can be used in order to obtain solutions more effectively. As an example, in Chapter 3, we showed that logic PBINT characterizes exactly the set of arithmetical search problems that can be recognized in NP. Thus, if a problem is specified within the syntax of logic PBINT and in a knowledge representation language that contains PBINT, it can be automatically deduced that the specified problem can be solved through a uniform polynomial time translation into a SAT problem (even if the language is capable of specifying more computationally complex). Moreover, our proofs in Chapter 3, when interpreted constructively, provide one such uniform translation.

Future Direction: Expressiveness of Other Built-in Constructs

Although Chapter 3 discusses built-in arithmetic in declarative KR programming, the benefits of our study is a consequence of the approach we have taken rather than the specific constructs (i.e., built-in arithmetic) or the specific complexity class (i.e., NP) we have chosen to study. Our choice to study built-in arithmetical constructs and complexity class NP was due to their importance in knowledge representation languages. However, many similar results can be derived for other types of constructs and other complexity classes. To mention a few, of particular interest to us, are the constructs to create and work with lists and sets. These constructs are very useful for naturally specifying many knowledge representation problems. Designing and implementing a declarative language with such built-in constructs that is also guaranteed to capture a complexity class such as NP is extremely useful because it allows users to describe their problems conveniently while guaranteeing effective solving process using an underlying solver.

Needless to say that the expressiveness and capturing properties should not and need not be limited to the complexity class NP. For example, the computational power of many new KR languages has gone beyond NP and we are now observing more and more languages that are designed to solve Σ_2^P-complete problems and even some languages that are designed to solve PSPACE-complete problems. Thus, it is important to study the expressiveness of the built-in constructs these languages offer.

Future Direction: Practical Language with Arithmetic

In Chapter 3, we showed that PBINT can express all arithmetical search problems that are recognizable in NP. A consequence of this approach is that adding new polytime computable arithmetical constructs does not increase PBINT's expressiveness because all such polytime computable operators can be axiomatized within PBINT. However, in a practical declarative language, such constructs are necessary for convenience of users of declarative languages. Aggregates are an example of the type of arithmetical constructs that can be axiomatized inside NP but are still needed in any practical declarative language because it is not convenient for a user to always have to axiomatize aggregates.

Hence, we deem it worthwhile to design a practical language that has the same guarantees as PBINT while providing as many syntactical sugars as possible. The design PBINT-Spec in Chapter 3 is a small step in this direction but a lot more is still needed.

7.2 Declarative and Modular Representation of Problems

In Chapter 4, we defined a framework to combine modules independently of the languages these modules are implemented in. The main motivation for our framework is to use the power of different declarative languages in a single framework. We introduced a new supported semantics for our modular systems in Chapter 5 and showed that supported semantics has many interesting implications such as the ability to more easily specify some interesting problems and the ability to generalize different semantics of multi-context systems. Motivated by these two properties, we extended supported semantics to multi-context systems in Chapter **??** and to propositional programs in Chapter 6.

7.2.1 Summary of Contributions

In the three Chapters of 3, **??** and 6, among other things, we did the following:

Three Semantics: Model-theoretical, Operational and Supported. We introduced three different semantics for our modular system framework: the model-theoretical semantics, the operational semantics and the supported semantics. The two former semantics are equivalent, i.e., the models accepted by the model-theoretical semantics is closely related to the denotation given by the operational semantics. Supported model semantics generalizes the two other semantics and coincides with them exactly when justifications are non-existent.

Expressiveness and Complexity. We studied the expressiveness and complexity of modular systems under its model-theoretical or operational semantics. We showed the robustness of our modular systems framework by showing that the expressiveness of modular systems remains invariant under varying complexity constraints on primitive modules.

Comparative Expressiveness Study. We showed that our modular system framework under supported model semantics is comparatively more expressive than multi-context systems because two different semantics of multi-context systems can be simultaneously translated into modular systems under supported semantics. Moreover, although empty justification and reducibility-based justifications are enough for expressing different semantics of multi-context systems, supported semantics allows us to use many other justification functions and therefore properly generalizes either of the two previous semantics for multi-context systems.

Lazy Solving Algorithm. Using the model-theoretical view on modular systems, we developed an algorithm to lazily solve modular systems. We showed the effectiveness of our algorithm by modelling three state-of-the-art systems that combine different reasoning methods and showing that our lazy solving algorithm works similar to these practical systems.

Approximation of Solutions. Using the operational view on modular systems, we developed two approximation methods for approximately solving modular systems that are represented by a positive or a negative loop. We showed that our approximation methods can be combined into our lazy solving algorithm to obtain a better solving procedure and to sometimes reduce the complexity of the solving procedure.

Supported Semantics for Other Systems. We introduced supported semantics to two previously defined languages for non-monotonic reasoning: the multi-context systems (Chapter **??**) and the propositional logic programs (Chapter 6). For multi-context systems, the addition of supported semantics allowed us to (1) unify equilibrium semantics and grounded equilibrium semantics, (2) lift the condition of reducibility from grounded equilibrium semantics and to

make this semantics applicable to all multi-context systems, and, (3) open the doors for multitudes of new semantics that could not have been characterized in terms of either the equilibrium or grounded equilibrium semantics but that could be characterized in terms of supported semantics. For propositional logic programs, the introduction of supported semantics created a natural extension of stable model semantics that preserves the interesting properties of minimality and rationality of stable models.

7.2.2 Future Direction: Implementation

One of the most important extensions to our work is to develop an efficient implementation of our lazy solving algorithm and to provide a framework in which all parties involved in a problem can easily develop specialized CCAV oracles for their specific modules. Moreover, we deem it useful to also implement several generic oracles for important declarative languages that can be readily used to combine these languages together.

7.2.3 Future Direction: New Approximation Methods

One of the advantages of our approach when compared to other paradigms for combined solving is the possibility of introducing powerful mechanisms to approximately find solutions of a modular system without using the underlying solver. We showcased this possibility by approximating monotone and anti-monotone feedbacks. However, the possibilities to advance this direction are not limited to these two cases.

One of the most promising possibilities to advance approximation methods to modular systems is using the algebraic methods developed by the authors of [51] to define an ultimate approximation for a non-monotonic operator.

Another approach for defining new approximation methods is to find interesting and general properties that can be utilized for intelligently pruning the search space of the underlying solver. One such property that has been proved to be useful in other areas is the convexity of the solutions of a modular system. It has been shown that defining a meaningful measure of convexity can hugely reduce the space needed to search for solutions.

7.2.4 Future Direction: Extended Semantics

We showed in Chapter 5 that extending the semantics of modular systems can make it more convenient for users to express their problems using modular systems and, thus, new extended semantics

can greatly increase the usability of modular systems. One of the possible extensions to the semantics of modular systems is the notion of preference. It seems possible to define preference in modlar systems in a way that generalizes supported semantics, i.e., there is a natural translation from supported semantics to preference semantics so that supported models of the original modular system are the most preferred models of the translates system. Such an extension to the semantics of modular systems would fill the gap that currently exists in between supported models and non-supported models.

Bibliography

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

[2] `http://www.cs.sfu.ca/~ter/eternovska/Software_files/Experiments_LPAR18.tar`.

[3] The Asparagus Library of Examples for ASP Programs, http://asparagus.cs.uni-potsdam.de/.

[4] A. Aavani. Translating Pseudo-Boolean constraints into CNF. *Theory and Applications of Satisfiability Testing (SAT)*, pages 357–359, 2011.

[5] A. Aavani, S. Tasharrofi, G. Ünel, E. Ternovska, and D. G. Mitchell. Speed-up techniques for negation in grounding. In *LPAR-16*, pages 13–26, 2010.

[6] A. Aavani, X. N. Wu, D. G. Mitchell, and E. Ternovska. Grounding Cardinality Constraints. *LPAR-16 short paper*, 2010.

[7] A. Aavani, X. N. Wu, S. Tasharrofi, E. Ternovska, and D. 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.

[8] A. Aavani, X. N. Wu, E. Ternovska, and D. G. Mitchell. Grounding formulas with complex terms. In *Canadian AI, the 24th Canadian Conference on Artificial Intelligence*, pages 13–25, 2011.

[9] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, 1985.

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

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

[12] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[13] K. R. Apt and R. N. Bol. Logic programming and negation: A survey. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1994.

[14] S. E.-D. Bairakdar, M. Dao-Tran, T. Eiter, M. Fink, and T. Krennwallner. The DMCS solver for distributed nonmonotonic multi-context systems. In T. Janhunen and I. 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.

[15] M. Balduccini. Modules and signature declarations for a-prolog: Progress report. In *Proceedings of the 1st International Workshop on Software Engineering for Answer Set Programming (SEA'07)*, pages 41–55, 2007.

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

[17] C. Baral, G. Brewka, and J. Schlipf, editors. *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 4483 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2007.

[18] C. Baral, J. Dzifcak, and H. Takahashi. Macros, macro calls and use of ensembles in modular answer set programming. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP'06)*, pages 376–390. Lecture Notes in Computer Science (LNCS), 2006.

[19] D. M. Barrington, N. Immerman, and H. Straubing. On uniformity within NC^1. *J. Comput. Syst. Sci.*, 41(3):274 – 306, 1990.

[20] S. Baselice, P. A. Bonatti, and M. Gelfond. Towards an integration of answer set and constraint solving. In *Proceedings of the 21st International Conference on Logic Programming (ICLP'05)*, pages 52–66, 2005.

[21] R. Baumgartner, G. Gottlob, and S. Flesca. Visual information extraction with Lixto. In *Proceedings of the 27th International Conference on Very Large Databases*, pages 119–128, Rome, Italy, September 2001. Morgan Kaufmann.

[22] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 283–293, 1992.

[23] M. Bögl, T. Eiter, M. Fink, and P. Schüller. The MCS-IE system for explaining inconsistency in multi-context systems. In T. Janhunen and I. 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.

[24] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, April–June 1985.

[25] G. Brewka and T. Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI'07) - Volume 1*, pages 385–390. AAAI Press, 2007.

[26] G. Brewka, T. Eiter, M. Fink, and A. Weinzierl. Managed multi-context systems. In T. Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 786–791. IJCAI/AAAI, 2011.

[27] S. R. Buss. *Bounded arithmetic*. PhD thesis, Princeton University, 1985.

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

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

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

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

[32] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, and T. Schaub. ASP-Core-2 input language format, 2012.

[33] B. D. Cat and M. Denecker. DPLL(Agg): An efficient smt module for aggregates. In *Proceedings of the 3rd International Workshop on Logic and Search (LaSh'10)*, 2010.

[34] R. Cerulli, P. Dell'Olmo, M. Gentili, and A. Raiconi. Heuristic approaches for the minimum labelling Hamiltonian cycle problem. *Electronic Notes in Discrete Mathematics*, 25:131–138, 2006.

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

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

[37] S. Cluet and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *ICDT '95: Proceedings of the 5th International Conference on Database Theory*, pages 54–67, London, UK, 1995. Springer-Verlag.

[38] A. Cobham. The intrinsic computational difficulty of functions. In *Y. Bar-Hillel ed., Proc. of the 1964 International Congress for Logic, Methodology, and the Philosophy of Science*, pages 24–30, 1964.

[39] A. Colmerauer. An introduction to prolog iii. *Communications of ACM*, 33(7):69–90, 1990.

[40] J. L. Connell and L. Shafer. *Structured rapid prototyping: an evolutionary approach to software development*. Yourdon Press, Upper Saddle River, NJ, USA, 1989.

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

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

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

[44] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter Section 29.3: The simplex algorithm, pages 790–804. MIT Press and McGraw-Hill, second edition, 2001.

[45] O. Coudert. Exact coloring of real-life graphs is easy. In *Design Automation Conference*, pages 121–126. IEEE COMPUTER SOCIETY, 1997.

[46] M. Dao-Tran, T. Eiter, M. Fink, and T. Krennwallner. Modular nonmonotonic logic programming revisited. In Hill and Warren [109], pages 145–159.

[47] A. M. Davis. Operational prototyping: A new development approach. *IEEE Software*, 9(5):70–78, Sept. 1992.

[48] M. J. G. de la Banda, K. Marriott, R. Rafeh, and M. Wallace. The modelling language Zinc. *Principles and Practice of Constraint Programming (CP 2006)*, pages 700–705, 2006.

[49] T. Dell'Armi, W. Faber, G. Ielpa, C. Koch, N. Leone, S. Perri, and G. Pfeifer. System description: DLV. In *LPNMR*, pages 424–428, 2001.

[50] M. Denecker. Extending classical logic with inductive definitions. In *Proc. CL'2000*, 2000.

[51] M. Denecker, V. W. Marek, and M. Truszczynski. Ultimate approximation and its application in nonmonotonic knowledge representation systems. *Inf. Comput.*, 192(1):84–121, 2004.

[52] M. Denecker and E. Ternovska. Inductive situation calculus. In *Proc., KR-04*, 2004.

[53] M. Denecker and E. Ternovska. A logic of non-monotone inductive definitions and its modularity properties. In *Proc., LPNMR-04*, 2004.

[54] M. Denecker and E. Ternovska. Inductive situation calculus. *Artificial Intelligence*, 2007.

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

[56] M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczyński. The second answer set programming competition. *LPNMR*, pages 637–654, 2009.

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

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

[59] N. Een and N. Sörensson. MiniSat v1.13 - a SAT solver with conflict-clause minimization, system description for the sat competition, 2005.

[60] T. Eiter, M. Fink, and P. Schüller. Approximations for explanations of inconsistency in partially known multi-context systems. In J. P. Delgrande and W. Faber, editors, *Proceedings of the 11th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR 2011), Vancouver, Canada*, volume 6645 of *Lecture Notes in Computer Science*, pages 107–119. Springer, 2011.

[61] T. Eiter, M. Fink, P. Schüller, and A. Weinzierl. Finding explanations of inconsistency in multi-context systems. In F. Lin, U. Sattler, and M. 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.

[62] T. Eiter, M. Fink, and S. Woltran. Semantical characterizations and complexity of equivalences in answer set programming. *ACM Trans. Comput. Logic*, 8, July 2007.

[63] T. Eiter, G. Gottlob, and H. Veith. Modular logic programming and generalized quantifiers. In *Proceedings of the 4th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR'97)*, pages 290–309. Springer-Verlag, 1997.

[64] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *Proceedings of the 19th International Joint Conference on Artificial intelligence (IJCAI'05)*, pages 90–96, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.

[65] C. Elkan. A rational reconstruction of nonmonotonic truth maintenance systems (research note). *Artif. Intell.*, 43(2):219–234, May 1990.

[66] O. Elkhatib, E. Pontelli, and T. C. Son. ASP-PROLOG: A system for reasoning about answer set programs in prolog. In *Proceedings of the 6th International Symposium on Practicl Aspects of Declarative Languages (PADL'04)*, pages 148–162, 2004.

[67] W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In J. J. Alferes and J. A. Leite, editors, *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, volume 3229 of *Lecture Notes in Computer Science*, pages 200–212. Springer, 2004.

[68] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. *Complexity of computation, SIAM-AMC proceedings*, 7:43–73, 1974.

[69] R. Fagin. Finite-model theory – a personal perspective. *Theoretical Comput. Sci.*, 116:3–31, 1993.

[70] P. Ferraris. Answer sets for propositional theories. In *Proceedings of the 8th international conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR'05, pages 119–131, Berlin, Heidelberg, 2005. Springer-Verlag.

[71] M. Fink, L. Ghionna, and A. Weinzierl. Relational information exchange and aggregation in multi-context systems. In J. P. Delgrande and W. 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 120–133. Springer, 2011.

[72] P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. *Logic Based Program Synthesis and Transformation*, pages 214–232, 2004.

[73] F. Focacci, A. Lodi, and M. Milano. Cost-based domain filtering. In J. Jaffar, editor, *Principles and Practice of Constraint Programming – CP'99*, volume 1713 of *Lecture Notes in Computer Science*, pages 189–203. Springer Berlin / Heidelberg, 1999.

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

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

[76] A. M. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, and I. Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13:268–306, 2008.

[77] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Commun. 24(2)*, pages 105–124, 2011.

[78] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental ASP solver. In *Proceedings of the 24th International Conference on Logic Programming (ICLP'08)*, volume 5366, pages 190–205. Lecture Notes in Computer Science (LNCS), 2008.

[79] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. *A User's Guide to gringo, clasp, clingo, and iclingo*, November 2008. http://potassco.sourceforge.net/.

[80] M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in *gringo* series 3. In J. Delgrande and W. Faber, editors, *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, volume 6645 of *Lecture Notes in Artificial Intelligence*, pages 345–351. Springer-Verlag, 2011.

[81] M. Gebser, B. Kaufmann, A. Neumann, and T. 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.

[82] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, Aug. 2012.

[83] M. Gebser, A. Konig, T. Schaub, S. Thiele, and P. Veber. The BioASP library: ASP solutions for systems biology. In *Proceedings of the 2010 22nd IEEE International Conference on Tools with Artificial Intelligence - Volume 01*, ICTAI '10, pages 383–389, Washington, DC, USA, 2010. IEEE Computer Society.

[84] M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In *Proceedings of the 25th International Conference on Logic Programming (ICLP'09)*, Lecture Notes in Computer Science (LNCS), pages 235–249. Springer-Verlag, 2009.

[85] M. Gebser and T. Schaub. Characterizing asp inferences by unit propagation. In *Proceedings of the 1st Workshop on Logic and Search Heuristics (LaSH'06), co-located with the 22nd International Conference on Logic Programming (ICLP'06)*, pages 41–56, 2006.

[86] M. Gebser, T. Schaub, and S. Thiele. Gringo : A new grounder for answer set programming. In C. Baral, G. Brewka, and J. S. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271. Springer, 2007.

[87] M. Gebser, T. Schaub, and S. Thiele. Gringo: A new grounder for answer set programming. In Baral et al. [17], pages 266–271.

[88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceeding of the 5th International Conference and Symposium on Logic Programming (ICLP/SLP'88)*, pages 1070–1080, 1988.

[89] M. Gelfond and V. Lifschitz. Logic programming. In D. H. Warren and P. Szeredi, editors, *Proceedings of the 6th International Conference on Logic Programming (ICLP)*, chapter Logic programs with classical negation, pages 579–597. MIT Press, Cambridge, MA, USA, 1990.

[90] G. Gottlob. Complexity results for nonmonotonic logics. *Journal of Logic and Computation*, 2(3):397–425, June 1992.

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

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

[93] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64(3):579–627, May 2002.

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

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

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

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

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

[99] E. Grädel. *Finite Model Theory and Descriptive Complexity*, pages 125–230. Springer, 2007.

[100] E. Grädel and Y. Gurevich. Metafinite model theory. *Inf. Comput.*, 140(1):26–81, 1998.

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

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

[103] E. Grädel and S. Kreutzer. Descriptive complexity theory for constraint databases. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic, CSL '99, Madrid*, volume 1683 of *LNCS*, pages 67–81. Springer, 1999.

[104] E. Grädel and K. Meer. Descriptive complexity theory over the real numbers. *Mathematics of Numerical Analysis: Real Number Algorithms*, 32:381–403, 1996.

[105] E. Grädel and K. Meer. Descriptive complexity theory over the real numbers. In J. Renegar, M. Shub, and S. Smale, editors, *Mathemetics of Numerical Analysis: Real Number Algorithms*, number 32 in Lectures in Applied Mathemetics, pages 381–403. AMS, 1996.

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

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

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

[109] P. M. Hill and D. S. 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.

[110] W. Hodges. Cambridge University Press, 1993.

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

[112] J. N. Hooker. Logic, optimization, and constraint programming. *INFORMS Journal on Computing*, 14(4):295–321, Oct. 2002.

[113] J. N. Hooker. *Integrated Methods for Optimization*. International Series in Operations Research & Management Science. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[114] J. N. Hooker. Planning and scheduling by logic-based benders decomposition. *Journal of Operations Research*, 55(3):588–602, May 2007.

[115] N. Immerman. Relational queries computable in polynomial time. In *STOC '82: Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 147–152, 1982.

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

[117] N. Immerman. Relational queries computable in polytime. 68:86 –104, 1986.

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

[119] T. Janhunen, E. Oikarinen, H. Tompits, and S. Woltran. Modularity aspects of disjunctive stable models. In *Proceedings of the 9th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR'07)*, volume 4483, pages 175–187. Lecture Notes in Artificial Intelligence (LNAI), 2007.

[120] M. Järvisalo, E. Oikarinen, T. Janhunen, and I. Niemelä. A module-based framework for multi-language constraint modeling. In *Proceedings of the 10th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR'09)*, volume 5753 of *Lecture Notes in Computer Science (LNCS)*, pages 155–168. Springer-Verlag, 2009.

[121] S. C. Kleene. *Introduction to Mathematics*, page 492. New York, D. Van Nostrand Company, Inc., 1952.

[122] P. G. Kolaitis and C. H. Papadimitriou. Why not negation by fixpoint? In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'88)*, pages 231–239, New York, NY, USA, 1988. ACM.

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

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

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

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

[127] R. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM Journal on Computing*, 6(3):467–480, 1977.

[128] D. Leivant. A foundational delineation of computational feasibility. In *LICS '91: Proceedings of the sixth Annual IEEE Symposium on Logic in Computer Science*, pages 2–11, 1991.

[129] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[130] H. Levesque. Foundations of a functional approach to knowledge representation. *Artificial Intelligence*, 23(2):155–212, July 1984.

[131] H. J. Levesque. A logic of implicit and explicit belief. In *Proceedings of the Fourth National Conference on Artificial Intelligence*, pages 198–202, Austin, Texas, August 1984. American Association for Artificial Intelligence.

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

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

[134] V. Lifschitz. Thirteen definitions of a stable model. In A. Blass, N. Dershowitz, and W. Reisig, editors, *Fields of logic and computation*, pages 488–503. Springer-Verlag, Berlin, Heidelberg, 2010.

[135] V. Lifschitz and H. Turner. Splitting a logic program. In *Proceedings of the 11th International Conference on Logic Programming (ICLP'94)*, pages 23–37, Cambridge, MA, USA, 1994. MIT Press.

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

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

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

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

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

[141] M. Mariën, J. Wittocx, M. Denecker, and M. Bruynooghe. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In H. Kleine Büning and X. Zhao, editors, *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, volume 4996 of *Lecture Notes in Computer Science*, chapter 20, pages 211–224. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[142] J. P. Marques-silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[143] V. S. Mellarkod, M. Gelfond, and Y. Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53(1):251–287, 2008.

[144] S. J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[145] K. P. Mikkilineni and S. Y. W. Su. An evaluation of relational join algorithms in a pipelined query processing environment. *IEEE Trans. Softw. Eng.*, 14(6):838–848, 1988.

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

[147] D. G. Mitchell. A SAT solver primer. *Bulletin of the European Association on Theoretical Computer Science (EATCS)*, 85:112–132, 2005.

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

[149] D. G. Mitchell and E. Ternovska. A framework for representing and solving NP search problems. In *Proc. AAAI'05*, pages 430–435, 2005.

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

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

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

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

[154] R. C. Moore. Semantical considerations on nonmonotonic logic. *Artif. Intell.*, 25(1):75–94, Jan. 1985.

[155] B. Nebel. On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research*, 12:271–315, 2000.

[156] I. Niemelä. Integrating answer set programming and satisfiability modulo theories. In *Proceedings of the 10th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR'09)*, volume 5753 of *Lecture Notes in Computer Science (LNCS)*, pages 3–3. Springer-Verlag, 2009.

[157] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. In F. Baader and A. Voronkov, editors, *Proceedings of 11th International Conferemce on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'04)*, Lecture Notes in Computer Science (LNCS), pages 36–50. Springer, March 2004.

[158] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving sat and sat modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM (JACM)*, 53:937–977, November 2006.

[159] E. Oikarinen. Modular answer set programming. In *Proc. of ICLP'07*, pages 462–463. Springer-Verlag, 2007.

[160] E. Oikarinen and T. Janhunen. Modular equivalence for normal logic programs. In *Proceedings of 11th Workshop on Non-monotonic Reasoning (NMR'06)*, pages 10–18, 2006.

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

[162] P. M. Pardalos and M. G. C. Resende. *Handbook of Applied Optimization*, volume 126. Oxford University Press, New York, 2002.

[163] M. Patterson, Y. Liu, E. Ternovska, and A. Gupta. Grounding for model expansion in k-guarded formulas with inductive definitions. In *Proc. IJCAI'07*, pages 161–166, 2007.

[164] D. Pearce. Equilibrium logic. *Annals of Mathematics and Artificial Intelligence*, 47(1-2):3–41, 2006.

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

[166] J. Renegar, M. Shub, and S. Smale, editors. *Mathemetics of Numerical Analysis: Real Number Algorithms*. Number 32 in Lectures in Applied Mathemetics. AMS, 1996.

[167] F. Ricca and N. Leone. Disjunctive logic programming with types and objects: The DLV+ system. *Journal of Applied Logic*, 5(3):545 – 573, 2007.

[168] F. Rossi, P. v. Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[169] W. Scheufele and G. Moerkotte. On the complexity of generating optimal plans with cross products. In *In PODS Conference*, pages 238–248, 1997.

[170] R. Sebastiani. Lazy satisfiability modulo theories. *Jurnal of Satisfiability, Boolean Modeling and Computation (JSAT)*, 3:141–224, 2007.

[171] Y.-D. Shen. Well-supported semantics for description logic programs. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pages 1081–1086. AAAI Press, 2011.

[172] Y.-D. Shen and K. Wang. Extending logic programs with description logic expressions for the semantic web. In *Proceedings of the 10th international conference on The semantic web - Volume Part I*, ISWC'11, pages 633–648, Berlin, Heidelberg, 2011. Springer-Verlag.

[173] Y.-D. Shen and K. Wang. FLP semantics without circular justifications for general logic programs. In J. Hoffmann and B. Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*. AAAI Press, 2012.

[174] Y.-D. Shen and J.-H. You. A default approach to semantics of logic programs with constraint atoms. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR '09, pages 277–289, Berlin, Heidelberg, 2009. Springer-Verlag.

[175] S. Shlaer and S. J. Mellor. *Object-oriented systems analysis: modeling the world in data*. Yourdon Press, Upper Saddle River, NJ, USA, 1988.

[176] A. Skelley. *Theories and Proof Systems for PSPACE and the EXP-Time Hierarchy*. PhD thesis, University of Toronto, 2005.

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

[178] L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Comput. Sci.*, 3:1–22, 1977.

[179] D. Suciu. Domain-independent queries on databases with external functions. *Theor. Comput. Sci.*, 190(2):279–315, 1998.

[180] V. Svejdar. On the polynomial-space completeness of intuitionistic propositional logic. *Arch. Math. Log.*, 42(7):711–716, 2003.

[181] T. Swift and D. S. Warren. *The XSB System*, 2009.

[182] T. Syrjänen. *Lparse 1.0 User's Manual*, 2000. http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz.

[183] L. Tari, C. Baral, and S. Anwar. A language for modular answer set programming: Application to acc tournament scheduling. In *Proc. of ASP'05*, CEUR-WS, pages 277–292, 2005.

[184] S. Tasharrofi. A rational extension of stable model semantics to the full propositional language. In F. Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 1118 – 1124. IJCAI, AAAI/IJCAI, August 2013.

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

[186] S. Tasharrofi and E. Ternovska. PBINT, a logic for modelling search problems involving arithmetic. In *Proceedings of the 17th Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'17)*. Springer, October 2010. LNCS 6397.

[187] S. Tasharrofi and E. Ternovska. A semantic account for modularity in multi-language modelling of search problems. In *Proceedings of the 8th International Symposium on Frontiers of Combining Systems (FroCoS)*, pages 259–274, October 2011.

[188] S. Tasharrofi, X. N. Wu, and E. Ternovska. Solving modular model expansion tasks. In *Proceedings of the 25th International Workshop on Logic Programming (WLP'11)*, volume abs/1109.0583. Computing Research Repository (CoRR), September 2011.

[189] E. Ternovska and D. G. Mitchell. Declarative programming of search problems with built-in arithmetic. In *Proc. of 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 942–947, 2009.

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

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

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

[193] R. Topor. Safe database queries with arithmetic relations. In *Proc. 14th Australian Computer Science Conf*, pages 1–13, 1991.

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

[195] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. 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.

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

[197] M. Truszczyński. Reducts of propositional theories, satisfiability relations, and generalizations of semantics of logic programs. *Artif. Intell.*, 174(16-17):1285–1306, Nov. 2010.

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

[199] H. Turner. Splitting a default theory. In *Proceedings of the 13th National Conference on Artificial intelligence (AAAI'96) - Volume 1*, pages 645–651. AAAI Press, 1996.

[200] P. Vaezipoor, D. Mitchell, and M. Mariën. Lifted unit propagation for effective grounding. In *Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP'11)*, volume abs/1109.1317. Computing Research Repository (CoRR), 2011.

[201] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)*, 23(4):733–742, Oct. 1976.

[202] A. Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and Systems Sciences*, 47(1):185–221, Aug. 1993.

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

[204] M. Vardi. Complexity of relational query languages. 68:137 –146, 1986.

[205] M. Y. Vardi. The complexity of relational query language. In *14th ACM Symp.on Theory of Computing, Springer Verlag (Heidelberg, FRG and NewYork NY, USA)-Verlag*, 1982.

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

[207] J. Vennekens, D. Gilis, and M. Denecker. Splitting an operator: Algebraic modularity results for logics with fixpoint semantics. *ACM Transactions on Computational Logic*, 7(4):765–797, 2006.

[208] J. Wittocx. Finite domain and symbolic inference methods for extensions of first-order logic. *AI Communications*, 24(1):91–93, Jan. 2011.

[209] J. Wittocx and M. Marien. *The IDP System*. KU Leuven, www.cs.kuleuven.be/~dtai/krr/software/idpmanual.pdf, June 2008.

[210] J. Wittocx and M. Marien. *The IDP System*. KUL, August 2009. http://www.cs.kuleuven.be/~dtai/krr/software/idpmanual.pdf.

[211] J. Wittocx, M. Mariën, and M. Denecker. Grounding with bounds. In *Proc. AAAI'08*, pages 572–577, 2008.

[212] J. Wittocx, M. Marién, and M. 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.

[213] J. Wittocx, M. Mariën, and M. Denecker. Grounding FO and FO(ID) with bounds. *Journal of Artificial Intelligence Research (JAIR)*, 38(1):223–269, May 2010.

[214] X. N. Wu. Solving model expansion tasks: System design and modularity. Master's thesis, Simon Fraser University, Burnaby, BC, Canada, August 2012.

[215] X. N. Wu and L. Swanson. *The Enfragmo System*, 2011. `http://www.cs.sfu.ca/~ter/eternovska/Software_files/Enfragmo-man.pdf`.

[216] T. Yato and T. Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Trans Fundam Electron Commun Comput Sci (Inst Electron Inf Commun Eng)*, E86-A(5):1052–1060, 2003.