TKK Dissertations in Information and Computer Science
Espoo 2008 TKK-ICS-D7

MODULARITY IN ANSWER SET PROGRAMS

Emilia Oikarinen

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Faculty of Information and Natural Sciences, for public examination and debate in Auditorium T2 at Helsinki University of Technology (Espoo, Finland) on the 24 of October, 2008, at 12 o'clock noon.

Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science

Teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietojenkäsittelytieteen laitos

Distribution:
Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science
P.O.Box 5400
FI-02015 TKK
FINLAND
URL: http://ics.tkk.fi
Tel. +358 9 451 1
Fax +358 9 451 3369
E-mail: series@ics.tkk.fi

ISBN 978-951-22-9581-4 (Print)
ISBN 978-951-22-9582-1 (Online)
ISSN 1797-5050 (Print)
ISSN 1797-5069 (Online)
URL: http://lib.tkk.fi/Diss/2008/isbn9789512295821/

Multiprint Oy
Espoo 2008

ABSTRACT: Answer set programming (ASP) is an approach to rule-based constraint programming allowing flexible knowledge representation in variety of application areas. The declarative nature of ASP is reflected in problem solving. First, a programmer writes down a logic program the answer sets of which correspond to the solutions of the problem. The answer sets of the program are then computed using a special purpose search engine, an ASP solver. The development of efficient ASP solvers has enabled the use of answer set programming in various application domains such as planning, product configuration, computer aided verification, and bioinformatics.

The topic of this thesis is modularity in answer set programming. While modern programming languages typically provide means to exploit modularity in a number of ways to govern the complexity of programs and their development process, relatively little attention has been paid to modularity in ASP. When designing a module architecture for ASP, it is essential to establish full compositionality of the semantics with respect to the module system. A balance is sought between introducing restrictions that guarantee the compositionality of the semantics and enforce a good programming style in ASP, and avoiding restrictions on the module hierarchy for the sake of flexibility of knowledge representation.

To justify a replacement of a module with another, that is, to be able to guarantee that changes made on the level of modules do not alter the semantics of the program when seen as an entity, a notion of equivalence for modules is provided. In close connection with the development of the compositional module architecture, a transformation from verification of equivalence to search for answer sets is developed. The translation-based approach makes it unnecessary to develop a dedicated tool for the equivalence verification task by allowing the direct use of existing ASP solvers.

Translations and transformations between different problems, program classes, and formalisms are another central theme in the thesis. To guarantee efficiency and soundness of the translation-based approach, certain syntactical and semantical properties of transformations are desirable, in terms of translation time, solution correspondence between the original and the transformed problem, and locality/globality of a particular transformation.

In certain cases a more refined notion of minimality than that inherent in ASP can make program encodings more intuitive. Lifschitz' parallel and prioritized circumscription offer a solution in which certain atoms are allowed to vary or to have fixed values while others are falsified as far as possible according to priority classes. In this thesis a linear and faithful transformation embedding parallel and prioritized circumscription into ASP is provided. This enhances the knowledge representation capabilities of answer set programming by allowing the use of existing ASP solvers for computing parallel and prioritized circumscription.

KEYWORDS: modular program development, module theorem, stable models, compositional semantics, equivalence verification, congruence relations, parallel and prioritized circumscription, faithful transformations, answer set programming, nonmonotonic reasoning

TIIVISTELMÄ: Vastausjoukko-ohjelmointi (answer set programming, ASP) on rajoiteohjelmoinnin osa-alue, jonka sääntöpohjaisen kielen ilmaisuvoima mahdollistaa monien sovellusongelmien joustavan ja luontevan ratkaisemisen. Ongelmanratkaisun vaiheet tuovat esiin ASP:n deklaratiivisen luonteen. Ensin kirjoitetaan logiikkaohjelma, eli joukko sääntöjä, jonka stabiilit mallit vastaavat ongelman ratkaisuja. Tämän jälkeen ohjelman stabiilit mallit lasketaan käyttäen erityistä ASP-ratkaisinta. Tehokkaiden ASP-ratkaisimien kehittäminen on mahdollistanut ASP:n hyödyntämisen muun muassa suunnitteluongelmien ratkaisemisessa ja tietokoneavusteisessa verifioinnissa.

Tässä väitöskirjassa tarkastellaan modulaarisuutta ASP:ssa. Toisin kuin perinteisessä ohjelmoinnissa, jossa ohjelmakoodin uudelleenkäyttö ja ohjelmien jakaminen pienempiin osiin eli moduuleihin on ollut mahdollista jo pitkään, ASP-ohjelma nähdään tyypillisesti kokonaisuutena. Kun ohjelmien koko kasvaa ja niistä tulee monimutkaisempia, on yhä ilmeisempää, että myös ASP-ohjelmointi hyötyisi vastaavanlaisista ohjelmakehitystä tukevista mekanismeista. Tärkeä kriteeri moduulijärjestelmän suunnittelussa on saavuttaa täysi yhteensopivuus stabiilien mallien semantiikan kanssa niin, että moduulien stabiileista malleista voidaan yhdistellä koko ohjelman stabiili malli ja päinvastoin. Jotta tämä tavoite saavutetaan, ei voida sallia mielivaltaisten moduulien yhdistämistä. Pyrkimyksenä on löytää tasapaino rajoitteille niin, että ne samalla kannustavat hyvään ohjelmointitapaan ja edelleen mahdollistavat joustavan tietämyksen esittämisen moduuleilla.

Ekvivalenssin käsitettä tarvitaan, jotta voidaan perustellusti korvata alimoduuli toisella laajemmassa kokonaisuudessa. Työssä esitelty modulaarinen ekvivalenssi on kongruenssi moduulien yhdistämisen suhteen, mikä takaa, että modulaarisesti ekvivalentit moduulit säilyttävät semantiikan missä tahansa yhdistämisoperaation sallimassa kontekstissa. Jotta ekvivalenssin tarkastamiseen ei tarvitsisi kehittää erityistä työkalua, esitellään muunnosfunktioon pohjautuva menetelmä, joka muuntaa ekvivalenssin tarkastuksen stabiilin mallin löytämisen ongelmaksi mahdollistaen näin ASP-ratkaisinten käytön. Tällainen muunnoksiin pohjautuva lähestymistapa mahdollistaakin olemassaolevan ASP:n teorian ja työkalujen tehokkaan hyödyntämisen ja muunnokset eri ongelmien ja ohjelmaluokkien välillä nousevat yhdeksi työn keskeiseksi teemaksi. Muunnosten tehokkuutta analysoidaan niiden aikavaativuuden, mallivastaavuuden säilyttämisen ja modulaarisuuden suhteen.

On myös sovelluksia, joissa stabiilien mallien minimaalisuus tekee ongelman logiikkaohjelmaesityksen muodostamisesta haastavaa. Lifschitz on esittänyt minimimallikäsitteen, joka tuo ratkaisun tällaisiin tilanteisiin. Kaikkien atomien minimoinnin sijaan määritellään, että joidenkin atomien totuusarvot varioivat vapaasti, osan totuusarvot on kiinnitetty ja loput minimoidaan mahdollisesti eri prioriteeteilla. Tämä minimimallipäättely voidaan upottaa stabiilien mallien semantiikkaan työssä esitettyä muunnosta käyttäen. Muunnoksen lineaarisuus ja tarkka mallivastaavuus mahdollistavat, että ASP:n ilmaisuvoimaa voidaan laajentaa hyödyntäen yhä nykyisiä ASP-ratkaisimia.

AVAINSANAT: modulaarinen ohjelmakehitys, stabiilit mallit, kompositionaalinen semantiikka, ekvivalenssin verifiointi, kongruenssirelaatio, varioitavat atomit minimimalleissa, mallivastaavuuden säilyttävät muunnokset, vastausjoukko-ohjelmointi, epämonotoninen päättely

CONTENTS

PREFACE

Soundtrack for this thesis: Rammstein (especially Mutter) and Johnny Cash (American III–V). Great music.

This thesis is a result of research carried out in Laboratory for Theoretical Computer Science (Department of Information and Computer Science starting from January 2008) at Helsinki University of Technology. The research has been funded by Helsinki Graduate School in Computer Science and Engineering (HeCSE), and Academy of Finland under projects #211025 and #122399. The personal grants from Nokia Foundation, Finnish Foundation for Technology Promotion TES, Emil Aaltonen foundation, and Kainuu Foundation of Finnish Cultural Foundation are gratefully acknowledged.

I am deeply indebted to my supervisor Professor Ilkka Niemelä. He is an excellent teacher and mentor, and I highly value all that I have learned from him during these years. I thank him also for providing me the opportunity to concentrate to this work. I am grateful to my instructor Docent Tomi Janhunen for always finding the time to discuss new ideas, problems encountered and ways to overcome them. In addition to providing guidance and comments during the whole process, his contribution to the articles in this thesis is significant. I also would like to thank my co-authors Dr. Stefan Woltran and Dr. Hans Tompits for their contributions.

I thank the pre-examiners Professor Hudson Turner (University of Minnesota Duluth) and Professor Katsumi Inoue (National Institute of Informatics, Japan) for their comments and suggestions, and I am grateful to Professor Michael Gelfond (Texas Tech University) for the honor of having him as the opponent.

I thank all my colleagues for creating an inspiring and enthusiastic working atmosphere. Especially the hard core coffee drinkers who meet after lunch at the coffee lounge, and my next-office-neighbor Dr. Tommi Junttila have brightened up the days many, many times.

Last but not least, I wish to thank my family, loved-ones, and friends. Without you, everything would mean nothing.

Helsinki, October 2008

Emilia Oikarinen

1 INTRODUCTION

In 1988, Gelfond and Lifschitz presented their seminal work titled "The stable model semantics for logic programming" [53] which introduced a novel declarative semantics for logic programs with negation. In twenty years the field which later became known as *answer set programming* (ASP) [98, 93, 52, 7, 128], has developed into an active research community with roots in declarative programing [69] and nonmonotonic logics [108], and with close connections to other fields in knowledge representation and reasoning [128, 110], such as, propositional satisfiability (SAT) solving, description logics, and constraint programming.

In answer set programming the problem at hand is solved declaratively

1. by writing down a logic program the answer sets of which correspond to the solutions of the problem, and

2. by computing the answer sets of the program using a special purpose search engine designed for this task.

The need for efficient *search engines* or *solvers* for computing answer sets was realized early on, and already after the mid-1990's ASP solvers SMODELS [118] and DLV [72] enabled the successful use of answer set programming in solving various knowledge representation and reasoning tasks. Application areas of ASP nowadays include, for example, planning [75], product configuration [119, 120], computer aided verification [58], wire routing in VLSI design [42], logical cryptanalysis [59, 1], network security [5], building a decision support system of NASA space shuttle [6], bioinformatics [8], and modelling the evolution of natural languages [41, 17].

The stable model semantics was originally presented for *normal logic programs* [53] but soon after a generalization for *disjunctive logic programs* was proposed [55, 56, 107]. Nowadays, numerous extensions to the underlying reasoning language have been introduced, motivated by a need to enhance knowledge representation capabilities in applications domains, [118, 79, 68, 50, 28, 106, 46, 121, 91, 113, 14, 27]. Moreover, there has been a growing effort to develop even more efficient ASP solving techniques and algorithms, resulting in a number of new ASP solvers, see [65, 82, 73, 85, 3, 49, 83], for instance.

The topic of this thesis is modularity in answer set programming. Looking from a software engineering point of view, it is desirable if not even necessary in the process of developing and maintaining problem encodings, that there is a *compositional* approach to answer set programming which allows composing programs from *modules* of some kind. In the thesis classes of SMODELS *programs* and *disjunctive logic programs* are considered. These classes can be seen as two representatives of expressive program classes; the former is used to solve problems on the first level of polynomial hierarchy, and the latter on the second level of polynomial hierarchy. Logic programs are assumed to be *finite* and *propositional*. However, as a rule with variables is typically seen as a shorthand for all its ground instantiations with respect to the *Herbrand universe* of the program in ASP, the theory developed in

this thesis can be extended to handle logic programs with variables, too. The finiteness assumption could be dropped in many places, but this has a potential of complicating definitions, and, for instance, with respect to the computational complexity, finiteness of programs is essential. Further restrictions are applied, for instance, in module interfaces and module compositions, when motivated by a "good programming style" in answer set programming or needed to keep computational complexity on a reasonable level.

In order to exploit the existing theory and tools in answer set programming as much as possible both on theoretical and on practical aspects, *translations* and *transformations* between different problems, program classes, and formalisms form a central theme in this work. To guarantee efficiency and soundness of this approach, certain syntactical and semantical properties of transformations are desirable, presented in terms of the translation time, maintenance of (exact) solution correspondence between the original and the transformed problem, and locality/globality of the particular transformation, which naturally intertwines with the theory developed in this thesis.

Modern programming languages typically provide means to exploit *modularity* in a number of ways to govern the complexity of programs and their development process. Indeed, the use of program modules or *objects* can be viewed as an example of the *divide-and-conquer* principle in the art of programming. The benefits of modular program development are numerous. A software system is much easier to design as a set of interacting components rather than a monolithic system with unclear internal structure. A modular design is more easily implementable as programming tasks can be delegated amongst a team of programmers. It also enables the re-use of code organized as module libraries, for instance.

Although modularity has been studied extensively in the context of *conventional logic programs*, for example, [20, 103, 90, 16, 48, 95, 57, 43, 88, 122, 25], relatively little attention has been paid to modularity in answer set programming. However, the problem encodings in ASP usually consist of at least two parts illustrating the *generate-and-test* (or *guess-and-check*) principle [132, 37, 19, 30, 93, 98, 75, 76] underlying in answer set programming. A generic problem specification part is used to test an input produced by a generator part which may vary according to a specific instance. However, the stable model semantics does not lend itself directly for program composition. The problem is that in general, stable models associated with parts of a program do not determine stable models assigned to their union. Such indivisibility of programs is likely to create problems as program instances tend to grow along the demands of new application areas of ASP emerging in semantic web, bioinformatics, and logical cryptanalysis, for instance.

There are two main criteria for the design of the module architecture adopted in this work. First of all, it is essential to establish the full compositionality of answer set semantics with respect to the module system. Second, a balance is sought between introducing restrictions that enforce a good programming style, and avoiding restrictions on the module hierarchy for the sake of flexibility of knowledge representation. The approaches to modularity in answer set programming proposed so far are based on a very strict syntactic conditions on the module hierarchy, for instance, by enforcing stratification of some kind, or by prohibiting recursion altogether [38, 123, 37,

9]. Approaches based on *splitting sets* [80, 38, 45] are satisfactory from the point of view of *compositional semantics*, but typically there is no explicit interface definition characterizing the interaction between modules. On the other hand, compositionality aspects are neglected altogether in syntactic approaches [60, 123] and this aspect of modelling remains completely at the programmer's responsibility.

Furthermore, a need for a suitable *equivalence relation* arises from the modular setting. It is necessary to be able to justify a replacement of a module with another, that is, to be able to guarantee that changes made on the level of modules do not alter the semantics of the program when seen as an entity. There are several notions of equivalence proposed for logic programs, for example in [78, 33, 40, 64, 102, 134]. For instance, if logic programs R_1 and R_2 have exactly the same answer sets, they are said to be *weakly/ordinarily equivalent* [78], denoted by $R_1 \equiv R_2$. Looking this from the answer set programming perspective, weakly equivalent programs produce the same solutions for the problem they formalize. If $R_1 \cup R \equiv R_2 \cup R$ for all programs R, then R_1 and R_2 are said to be *strongly equivalent* [78], denoted by $R_1 \equiv_s R_2$. Strongly equivalent programs preserve the solutions to the problem in every possible context in which they can be placed in.

Instead of designing a dedicated tool for the equivalence verification task, several approaches have been proposed to transform the problem of equivalence verification to one for which there exists already efficients solvers, such as, for instance, answer set programming, propositional satisfiability, and quantified Boolean formulas, [66, 126, 101, 133, 23, 125, 31]. However, these methods typically treat programs as integral entities which might limit the usefulness of the translation-based method in a setting where programs consist of modules. In this thesis the concept of equivalence for ASP is brought to module-level in close connection with the development of the compositional module architecture. A transformation from equivalence verification to search for stable models is developed accordingly, allowing this way the direct use of existing ASP solvers in verification tasks.

Even though the stable model semantics based on minimality leads to concise encodings of problems as programs, occasionally knowledge representation with answer set programming becomes complicated by the fact that all atoms appearing in a logic program are false by default. *Parallel and prioritized circumscription* [74] offer a more refined control of minimization in which certain atoms are allowed to *vary* or to have *fixed* values while others are falsified as far as possible. In the prioritized case *priority classes* for atoms being minimized are furthermore introduced. Unfortunately, in particular the varying atoms are not well-supported in answer set programming, although serious attempts to embed parallel/prioritized circumscription into disjunctive logic programming have been made [54, 112, 113, 130, 71, 131].

The last part of this thesis consists of an brief introduction to properties of translation functions in terms of the translation time, exact model correspondence, and modularity. An analysis of parallel circumscription reveals that even though the varying atoms are of rather global nature, it is possible to provide an efficient embedding for parallel and prioritized circumscription into answer set programming in terms of a *linear* and *faithful* translation function. This allows developing *modular* encodings using parallel and pri-

oritized circumscription for problems, which are otherwise hard to formalize as logic programs when all atoms are minimized. Thus, the knowledge representation capability of answer set programming is enhanced as the existing ASP solvers can be used to compute models for parallel and prioritized circumscription.

2 STRUCTURE OF THE THESIS

This thesis consists of a seven chapter summary and the following seven articles.

[**P1**] Emilia Oikarinen and Tomi Janhunen. Achieving compositionality of the stable model semantics for SMODELS programs. *Theory and Practice of Logic Programming*, to appear.

[**P2**] Tomi Janhunen, Emilia Oikarinen, Hans Tompits, and Stefan Woltran. Modularity aspects of disjunctive stable models. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, Proceedings of the 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 2007*, volume 4483 of *Lecture Notes in Artificial Intelligence*, pages 175–187. Springer.

[**P3**] Tomi Janhunen and Emilia Oikarinen. Automated verification of weak equivalence within the Smodels system. *Theory and Practice of Logic Programming*, 7(6):697–744, 2007.

[**P4**] Emilia Oikarinen and Tomi Janhunen. A translation-based approach to the verification of modular equivalence. *Journal of Logic and Computation*. Advance Access published, doi:10.1093/logcom/exn039, August 2008.

[**P5**] Tomi Janhunen and Emilia Oikarinen. Capturing parallel circumscription with disjunctive logic programs. In José Júlio Alferes and João Leite, editors, *Logics in Artificial Intelligence, Proceedings of the 9th European Conference, JELIA 2004, Lisbon, Portugal, September 2004*, volume 3229 of *Lecture Notes in Artificial Intelligence*, pages 134–146. Springer.

[**P6**] Emilia Oikarinen and Tomi Janhunen. CIRC2DLP — translating circumscription into disjunctive logic programming. In Chitta Baral, Gianluigi Grego, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning, Proceedings of the 8th International Conference, LPNMR 2005, Diamante, Italy, September 2005*, volume 3662 of *Lecture Notes in Artificial Intelligence*, pages 405–409. Springer.

[**P7**] Emilia Oikarinen and Tomi Janhunen. Implementing prioritized circumscription by computing disjunctive stable models. In Danail Dochev, Marco Pistore, and Paolo Traverso, editors, *Artificial Intelligence: Methodology, Systems, and Applications, Proceedings of the 13th International Conference, AIMSA 2008, Varna, Bulgaria, September 2008*, volume 5223 of *Lecture Notes in Artificial Intelligence*, pages 167–180. Springer.

The current chapter consists of a brief description of contents and contributions in this thesis. In Chapter 3 basic preliminaries of answer set programming and the stable model semantics are given. In Chapter 4 different

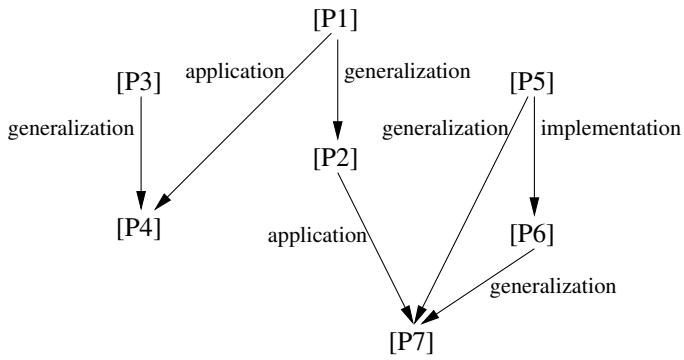

Figure 2.1: The relations between articles [**P1–P7**].

approaches to modularity in logic programming, and especially in answer set programming are discussed. Chapter 5 introduces a number of equivalence relations proposed for logic programs under the stable model semantics and methods for verification of equivalence. In Chapter 6 properties of translation functions between different program classes are discussed. As an illustrating example, a refined notion of minimality presented in terms of parallel and prioritized circumscription is embedded into answer set programming using a linear and faithful transformation. Chapter 7 presents some conclusions and topics for future work.

2.1 SUMMARY OF THE ARTICLES IN THE THESIS

The contents of the articles in this thesis are summarized in the following. In Chapters 4–6 the articles are placed in context with related work. The relations between the articles are summarized in Figure 2.1.

[**P1**] A module architecture for SMODELS programs is proposed. The architecture follows the ideas in [48] and introduces modules with a clearly defined input/output interface for communication. The stable model semantics is generalized for modules in a way that compositionality of the semantics under a module composition operation join is achieved. The compositionality property, which allows for computation of stable models separately for the parts of the program, is crystallized in a module theorem. Furthermore, a concept of modular equivalence is proposed. As necessary to allow replacements of modules, modular equivalence is a proper congruence relation for joins of modules.

[**P2**] The module architecture from [**P1**] is generalized to cover the class of disjunctive logic programs. It is demonstrated that the module theorem properly generalizes the splitting set theorem by Lifschitz and Turner [80]. Moreover, a general shifting principle is introduced as the generalization of a local shifting transformation by Eiter et al. [34] for program simplification. The modular equivalence relation has the congruence property in the disjunctive case, too.

[**P3**] A translation-based method for verifying weak equivalence [78] of two SMODELS programs is proposed. The idea is to transform the problem of verifying whether two logic programs are equivalent to the existence of stable models by introducing a linear translation function. This way ASP solvers can be directly utilized for equivalence verification. The method also covers the verification of visible equivalence [64] for programs having enough visible atoms. A more general class of weight constraint programs [118] is covered through a transformation into the class of SMODELS programs. The implementation of the translation shows promising performance compared to a naive method of cross-checking stable models.

[**P4**] The translation-based method for equivalence verification proposed in [**P3**] is modified to cover verification of modular equivalence of SMODELS program modules introduced in [**P1**]. The correctness proof of the translation demonstrates that the use of the module theorem [**P1**] leads to significant simplification compared to the proof given in [**P3**]. Experimental evaluation of the implementation suggests that the modularization of equivalence verification leads to potential time savings especially if the modules involved share a common context.

[**P5**] An expressiveness analysis shows that there is no linear, modular and faithful translation function from parallel circumscription to positive disjunctive logic programs. By relaxing the requirements, however, the first linear and faithful, though non-modular, translation from parallel circumscription [74] to disjunctive logic programs with negation is introduced.

[**P6**] An implementation of the translation-based method in [**P5**] is presented together with an experimental evaluation. Moreover, the implementation also covers prioritized circumscription by using Lifschitz' quadratic scheme [74] to embed prioritized case into parallel circumscription.

[**P7**] The translation from [**P5**] is generalized to directly cover prioritized circumscription by a linear and faithful transformation to disjunctive logic programming. The module theorem for disjunctive logic program modules in [**P2**] is used in arguing the faithfulness of the translation. The implementation of the linear translation clearly improves the quadratic one in [**P6**] and compares favorably to a tool by Wakaki and Tomita [131].

2.2 CONTRIBUTIONS OF THE AUTHOR

This section summarizes the contributions of the author to the articles constituting this thesis.

The article [**P1**] is co-authored by T. Janhunen. The key ideas and proofs are developed by the current author, and the paper is to a large extent written by the current author. The implementations and the experiments are the work of co-author.

The article [**P2**] is co-authored by T. Janhunen, H. Tompits, and S. Woltran. The current author is responsible for the proof of the congruence property of modular equivalence.

The article [**P3**] is co-authored by T. Janhunen who developed the original idea of the translation-based method for verification of equivalence and is responsible for the implementation of the method. The current author is responsible for the analysis of the computational complexity, the extension of the results to weight constraint programs and performing and reporting the experiments.

The article [**P4**] is co-authored by T. Janhunen. The paper is jointly written with the current author modifying the translation from [**P3**] to cover the verification of modular equivalence, and performing the experiments. The co-author is responsible for the implementation.

The article [**P5**] is co-authored by T. Janhunen. The paper is jointly written, with the co-author responsible for the expressiveness analysis and the initial idea of the translation for embedding parallel circumscription into disjunctive logic programming. The current author is responsible for the correctness proof of the translation.

The article [**P6**] is co-authored by T. Janhunen. The paper is jointly written, with the co-author responsible for designing the benchmark, and the current author responsible for the implementation and experimental evaluation.

The article [**P7**] is co-authored by T. Janhunen. The idea is jointly developed and the paper is jointly written. The current author is responsible for the correctness of the translation, its implementation, and performing the experimental evaluation.

3 ANSWER SET PROGRAMMING

In this chapter basic concepts of answer set programming are introduced starting from the syntax of logic programs and continuing with the semantics based on stable models.

3.1 LOGIC PROGRAMS

A *propositional logic program* is a finite set of *rules* which are expressions of the form

$$\mathsf{head} \leftarrow \mathsf{body}, \tag{3.1}$$

where the objects allowed to appear as the *head* of a rule or the *body* of a rule are limited by the class of logic programs under consideration. Intuitively the interpretation for a rule is that head must be satisfied, if body is satisfied. For example, a *normal or basic rule* is of the form

$$a \leftarrow b_1, \ldots, b_n, {\sim}c_1, \ldots, {\sim}c_m, \tag{3.2}$$

where a and each b_i, and c_j is a propositional atom, $n, m \in \mathbb{N}$, and the symbol $\sim$ denotes *negation as failure* [24] or *default negation* which differs from classical negation [55]. A *default literal* is either an atom a or its default negation ${\sim}a$. A *normal logic program* R is a finite set of basic rules. The set of atoms appearing in a program R is denoted by $\mathrm{At}(R)$. In the following, the rules in a program are separated with full stops and the symbol "$\leftarrow$" is omitted if the body of the rule is empty.

In this thesis the classes of SMODELS *programs* and *disjunctive logic programs* are of main interest. The rule types included in the language of SMODELS programs have been carefully chosen so that they enable knowledge representation in a compact form and are directly and efficiently implementable in the search engine of the SMODELS system [118][1]. The decision problem whether a program R has a stable model is **NP**-complete for both normal logic programs [92] and SMODELS programs (without optimization statements) [118]. Disjunctive logic programs enable representing problems in the second level of polynomial hierarchy, that is, deciding whether a disjunctive program R has a stable model is $\Sigma_2^{\mathbf{P}}$-complete [36]. Normal logic programs can be seen as a special case of both SMODELS programs and disjunctive logic programs. There is also a number of other interesting program classes, for instance, *weight constraint programs* [118] of which SMODELS programs can be viewed as a normal form, *extended disjunctive logic programs* [56, 107] which are disjunctive logic programs containing both default negation and classical negation, *nested logic programs* [79] which allow bodies and heads of rules contain arbitrary nested expressions, and logic programs with *aggregates* [68, 50, 28, 106, 46, 121, 91].

An SMODELS program R is a finite set of *basic constraint rules* [118]

[1] *Optimization statements* included in the input language of the SMODELS system are, however, not covered here.

which are either *weight rules* of the form

$$a \leftarrow w \leq \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}, \sim c_1 = w_{c_1}, \ldots, \sim c_m = w_{c_m}\} \quad (3.3)$$

or *choice rules* of the form

$$\{a_1, \ldots, a_h\} \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m \quad (3.4)$$

where a, a_i's, b_j's, and c_k's are atoms, $h > 0$, and $n, m \in \mathbb{N}$. In addition, a weight rule (3.3) involves a weight limit $w \in \mathbb{N}$ and the respective weights $w_{b_j} \in \mathbb{N}$ and $w_{c_k} \in \mathbb{N}$ associated with each *positive literal* b_j and *negative literal* $\sim c_k$. Weight rules (3.3) cover many other kinds of rules as their special cases, for instance, basic rules (3.2), *cardinality rules*

$$a \leftarrow l \leq \{b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m\}, \quad (3.5)$$

integrity constraints

$$\bot \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m, \quad (3.6)$$

and *compute statements*

$$\mathsf{compute}\ \{b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m\}. \quad (3.7)$$

A cardinality rule (3.5) is essentially a weight rule (3.3) where $w = l$ and all weights associated with literals equal to 1. A basic rule (3.2) is a special case of a cardinality rule (3.5) with $l = n + m$. The intuitive meaning of an integrity constraint (3.6) is that the conditions given in the body should never be simultaneously satisfied. The same can be stated using a basic rule

$$f \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m, \sim f \quad (3.8)$$

where f is a new atom dedicated to integrity constraints. Compute statements (3.7) effectively correspond to sets of integrity constraints $\bot \leftarrow \sim b_1$, $\ldots, \bot \leftarrow \sim b_n$ and $\bot \leftarrow c_1, \ldots, \bot \leftarrow c_m$. In the SMODELS system [118] the internal representation of programs is based on rules of the forms (3.2)–(3.5), and (3.7). Since the basic constraint rules provide a reasonable coverage of SMODELS programs, other rule types are viewed as syntactic sugar.

A disjunctive logic program R is a finite set of *disjunctive rules* of the form

$$a_1 \vee \ldots \vee a_h \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m, \quad (3.9)$$

where each a_i, b_j, and c_k is an atom, and $h > 0$, $n, m \in \mathbb{N}$. A basic rule (3.2) is a special case of a disjunctive rule (3.9) such that $h = 1$. It is also possible to allow disjunctive programs to contain integrity constraints (3.6), that is, to allow $h = 0$, but the same effect can be obtained using a rule of the form (3.8) similarly to the case of SMODELS programs.

The classes of SMODELS programs and disjunctive logic programs are denoted by $\mathcal{C}_\mathrm{s}$ and $\mathcal{C}_\mathrm{d}$, respectively. Moreover, $\mathcal{C} \in \{\mathcal{C}_\mathrm{s}, \mathcal{C}_\mathrm{d}\}$ is used to refer to a class of programs whenever it is not necessary to differentiate between $\mathcal{C}_\mathrm{s}$ and $\mathcal{C}_\mathrm{d}$. Overall, the goal is to unify the definitions presented in this thesis to cover both program classes, to the extent that it is possible. Finally, $\mathcal{C}_\mathrm{n}$ denotes the class of normal logic programs, and $\mathcal{C}_\mathrm{n} \subset \mathcal{C}_\mathrm{s}$ and $\mathcal{C}_\mathrm{n} \subset \mathcal{C}_\mathrm{d}$.

The set of atoms appearing in the heads of rules in $R \in \mathcal{C}$ is denoted by $\mathsf{head}(R)$ and the set of atoms appearing in the bodies of R by $\mathsf{body}(R)$. Furthermore, $\mathsf{body}^+(R)$ (respectively $\mathsf{body}^-(R)$) denotes the set of atoms having positive (respectively negative) body occurrences in R. With a slight abuse of notation $\mathsf{head}(r)$ is used to refer to the atoms appearing in the head of a rule r, that is, $\mathsf{head}(\{r\})$, and $\mathsf{body}(r)$, $\mathsf{body}^+(r)$, and $\mathsf{body}^-(r)$ are used in a similar way. If $\mathsf{body}(r) = \emptyset$, then r is a *fact*. An SMODELS program $R \in \mathcal{C}_\mathrm{s}$ is *positive* if each rule $r \in R$ is a weight rule (3.3) and $\mathsf{body}^-(R) = \emptyset$. A disjunctive logic program $R \in \mathcal{C}_\mathrm{d}$ is *positive* if $\mathsf{body}^-(R) = \emptyset$.

Example 3.1. Let $G = \langle N, E \rangle$ be a directed graph with n nodes, that is, $N = \{1, \ldots, n\}$ and $E \subseteq N \times N$. An atom $\mathsf{arc}(x, y)$ is used denote that there is a directed edge from node x to node y in G. The goal is to solve the Hamiltonian cycle problem for G, that is, to answer whether there is a cycle in the graph such that each node is visited exactly once returning to the starting node. To achieve this, a logic program $R_G \in \mathcal{C}_\mathrm{s}$ is designed as follows.

First, the graph G in question is encoded as a set of facts

$$\{\mathsf{arc}(x, y). \mid \langle x, y \rangle \in E\}. \tag{3.10}$$

An atom $\mathsf{hc}(x, y)$ is used to indicate that edge $\langle x, y \rangle$ is selected into a candidate for a Hamiltonian cycle. The selection is encoded by a set of choice rules

$$\{\{\mathsf{hc}(x, y)\} \leftarrow \mathsf{arc}(x, y). \mid \langle x, y \rangle \in N \times N\}. \tag{3.11}$$

Together with the constraint (3.16) enforcing c to be false in every model, the rules in (3.12) and (3.13) are used to guarantee that each node has exactly one outgoing edge, and the rules in (3.14) and (3.15) give the respective condition concerning incoming edges,

$$\begin{aligned}
&\{c \leftarrow 2 \leq \{\mathsf{hc}(x, 1), \ldots, \mathsf{hc}(x, n)\}. \mid x \in N\} && (3.12)\\
&\quad \cup\ \{c \leftarrow \sim\mathsf{hc}(x, 1), \ldots, \sim\mathsf{hc}(x, n). \mid x \in N\} && (3.13)\\
&\quad \cup\ \{c \leftarrow 2 \leq \{\mathsf{hc}(1, x), \ldots, \mathsf{hc}(n, x)\}. \mid x \in N\} && (3.14)\\
&\quad \cup\ \{c \leftarrow \sim\mathsf{hc}(1, x), \ldots, \sim\mathsf{hc}(n, x). \mid x \in N\} && (3.15)\\
&\quad \cup\ \{d \leftarrow c, \sim d.\}. && (3.16)
\end{aligned}$$

Finally, an atom $\mathsf{reached}(x)$ indicates that node x is reachable from the first node. In order to check that each node is reachable from the first node along the edges in the cycle, rules are introduced as follows,

$$\begin{aligned}
&\{\,\mathsf{reached}(y) \leftarrow \mathsf{hc}(1, y). \mid y \in N\} && (3.17)\\
&\ \cup\ \{\mathsf{reached}(y) \leftarrow \mathsf{reached}(x), \mathsf{hc}(x, y). \mid x \in N \setminus \{1\}, y \in N\} && (3.18)\\
&\ \cup\ \{e \leftarrow \sim e, \sim\mathsf{reached}(y). \mid y \in N\} && (3.19)
\end{aligned}$$

The logic program encoding R_G of the problem is the union of the rules in (3.10)–(3.19). ■

Given a logic program $R \in \mathcal{C}$ and atoms $a, b \in \mathrm{At}(R)$, a *depends directly* on b, denoted by $b \leq_1 a$, if and only if R contains a rule r such that $a \in$

$\mathsf{head}(r)$ and $b \in \mathsf{body}^+(r)$. The *positive dependency graph* of R, denoted by $\mathrm{Dep}^+(R)$, is a graph with $\mathrm{At}(R)$ and $\{\langle b, a\rangle \mid a \leq_1 b\}$ as the sets of nodes and edges, respectively. The reflexive and transitive closure of $\leq_1$ gives rise to the positive dependency relation $\leq$ over $\mathrm{At}(R)$. A *strongly connected component* (SCC) S of $\mathrm{Dep}^+(R)$ is a maximal set $S \subseteq \mathrm{At}(R)$ such that $b \leq a$ holds for every $a, b \in S$.

Given a logic program $R \in \mathcal{C}$, an *interpretation* M is a subset of $\mathrm{At}(R)$ defining which atoms in $\mathrm{At}(R)$ are true ($a \in M$) and which are false ($a \notin M$). The satisfaction relation $M \models r$ is defined for different types of rules r as follows.

Definition 3.2. *Given a logic program $R \in \mathcal{C}$ and an interpretation $M \subseteq \mathrm{At}(R)$,*

- *a weight rule (3.3) in R is satisfied in M if and only if*

$$w \leq \sum_{b \in \{b_1,\ldots,b_n\} \cap M} w_b + \sum_{c \in \{c_1,\ldots,c_m\} \setminus M} w_c$$

  implies $a \in M$,

- *a choice rule (3.4) in R is always satisfied in M, and*

- *a disjunctive rule (3.9) in R is satisfied in M if and only if $\{b_1, \ldots, b_n\} \subseteq M$ and $M \cap \{c_1, \ldots, c_m\} = \emptyset$ imply $M \cap \{a_1, \ldots, a_h\} \neq \emptyset$.*

An interpretation $M \subseteq \mathrm{At}(R)$ is a (classical) model of program $R \in \mathcal{C}$, denoted by $M \models R$, if and only if M satisfies all the rules in R. Classical models do not, however, correspond to the intuitive interpretation of rules seen as inference rules, since a rule is satisfied in M even if only its head it satisfied.

Example 3.3. Recall the program R_G in Example 3.1 designed to solve the Hamiltonian cycle problem for a directed graph G, the idea being that R_G has an answer set if and only if G has a Hamiltonian cycle.

Consider a graph $G_1 = \langle\{1, 2\}, \emptyset\rangle$, which has no Hamiltonian cycles. However, an interpretation

$$M = \{\mathsf{hc}(1, 2), \mathsf{hc}(2, 1), \mathsf{reached}(1), \mathsf{reached}(2)\} \subseteq \mathrm{At}(R_{G_1})$$

satisfies all the rules in R_{G_1}. The empty set of facts in (3.10) is clearly satisfied, and the choice rules in (3.11) are always satisfied by definition. The rules in (3.12)–(3.16) are satisfied, since $c \notin M$ and the bodies of the rules are not satisfied. Finally, since $\{\mathsf{reached}(1), \mathsf{reached}(2)\} \subseteq M$, the rules in (3.17)–(3.19) are satisfied. However, M does not represent a solution to the Hamiltonian cycle problem for G_1, and thus classical models do not match the intuition behind the rules. ■

A semantics based on *minimal models* of positive programs captures the intuition, however. A *minimal model* is a model for which there exists no proper subset that is also a model. More formally, a model $M \subseteq \mathrm{At}(R)$ of a positive logic program $R \in \mathcal{C}$ is minimal if and only if there is no $M' \models R$

such that $M' \subset M$. Thus, a minimal model of a program can be seen as a model that maximizes the falsity of atoms. The set of minimal models of a positive program $R \in \mathcal{C}$ is denoted by $\mathrm{MM}(R)$. Each positive program in $R \in \mathcal{C}_\mathrm{s}$ has a *unique minimal model* [118] (for $\mathcal{C}_\mathrm{n}$, see [86]), the *least model* of R, denoted by $\mathrm{LM}(R)$, whereas positive disjunctive programs may have several minimal models.

3.2 STABLE MODEL SEMANTICS

The *stable model semantics* [53] generalizes the minimal model semantics for programs containing negation. The key idea is to pre-evaluate the negations in the program with respect to a given model candidate and then compute the minimal models for the pre-evaluated program. The pre-evaluation preserves satisfiability: if $M \models R$, then M is also a model for the program obtained from R by pre-evaluating it with respect to M. A model candidate M for R is accepted as a *stable model* if it is a minimal model of the positive program obtained by pre-evaluating R with respect to M. The semantics for *normal* logic programs in terms of stable models was proposed by Gelfond and Lifschitz [53] and was later generalized for disjunctive logic programs [56, 107]. In [117] the stable model semantics is generalized for SMODELS programs. However, the reduced program is not explicitly present in the semantical definitions involving *deductive closures* [117]. An alternative definition adopted in [**P3**] can be viewed as a special case of the definition given in [118] covering weight constraint programs.

The *Gelfond-Lifschitz reduct* of program $R \in \mathcal{C}$ with respect to an interpretation $M \subseteq \mathrm{At}(R)$ is defined as follows.

Definition 3.4. *Given a logic program $R \in \mathcal{C}$ and an interpretation $M \subseteq \mathrm{At}(R)$, the reduct of R with respect to M, denoted by R^M, contains*

- *a rule $a \leftarrow b_1, \ldots, b_n$ if and only if there is a choice rule (3.4) in R such that $a \in M \cap \{a_1, \ldots, a_h\}$, and $M \cap \{c_1, \ldots, c_m\} = \emptyset$;*
- *a rule $a \leftarrow w' \leq \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}\}$ if and only if there is a weight rule (3.3) in R such that*

$$w' = \max(0, w - \sum_{c \in \{c_1, \ldots, c_m\} \setminus M} w_c); \text{ and}$$

- *a rule $a_1 \vee \ldots \vee a_h \leftarrow b_1, \ldots, b_n$ if and only if there is a disjunctive rule (3.9) in R such that $M \cap \{c_1, \ldots, c_m\} = \emptyset$.*

An interpretation $M \subseteq \mathrm{At}(R)$ is a *stable model* of a logic program $R \in \mathcal{C}$ if and only if $M \in \mathrm{MM}(R^M)$. The set of stable models of program $R \in \mathcal{C}$ is denoted by $\mathrm{SM}(R)$. Since positive SMODELS programs have a unique minimal model, the definition of stable models can alternatively be formulated for $\mathcal{C}_\mathrm{s}$: given a logic program $R \in \mathcal{C}_\mathrm{s}$, an interpretation $M \subseteq \mathrm{At}(R)$ is a stable model of R if and only if $M = \mathrm{LM}(R^M)$.

Example 3.5. Recall Example 3.3 demonstrating that the classical models of the encoding presented in Example 3.1 for graph $G_1 = \langle\{1,2\},\emptyset\rangle$ do not correspond to Hamiltonian cycles. Under the stable model semantics, however, the encoding works as intended. Consider, for instance, the interpretation

$$M = \{\mathsf{hc}(1,2), \mathsf{hc}(2,1), \mathsf{reached}(1), \mathsf{reached}(2)\} \subseteq \mathrm{At}(R_{G_1}).$$

The reduct $(R_{G_1})^M$ contains the following rules,

$$\begin{array}{ll}
\mathsf{hc}(1,1) \leftarrow \mathsf{arc}(1,1). & c \leftarrow 2 \leq \{\mathsf{hc}(1,1), \mathsf{hc}(1,2)\}.\\
\mathsf{hc}(1,2) \leftarrow \mathsf{arc}(1,2). & c \leftarrow 2 \leq \{\mathsf{hc}(2,1), \mathsf{hc}(2,2)\}.\\
\mathsf{hc}(2,1) \leftarrow \mathsf{arc}(2,1). & c \leftarrow 2 \leq \{\mathsf{hc}(1,1), \mathsf{hc}(2,1)\}.\\
\mathsf{hc}(2,2) \leftarrow \mathsf{arc}(2,2). & c \leftarrow 2 \leq \{\mathsf{hc}(1,2), \mathsf{hc}(2,2)\}.\\
d \leftarrow c. & \\
\mathsf{reached}(1) \leftarrow \mathsf{hc}(1,1). & \mathsf{reached}(2) \leftarrow \mathsf{hc}(1,2).\\
\mathsf{reached}(1) \leftarrow \mathsf{reached}(2), \mathsf{hc}(2,1). & \\
\mathsf{reached}(2) \leftarrow \mathsf{reached}(2), \mathsf{hc}(2,2). &
\end{array}$$

and has a unique minimal model, $\mathrm{LM}((R_{G_1})^M) = \emptyset$. Thus, M is not a stable model of R_{G_1}. In particular, $\mathrm{SM}(R_{G_1}) = \emptyset$, which captures the fact that graph G_1 has no Hamiltonian cycles. In general, given $M \in \mathrm{SM}(R_G)$, the projection $M \cap \{\mathsf{hc}(x,y) \mid \langle x,y\rangle \in N \times N\}$ corresponds to a Hamiltonian cycle in G, and there is a bijective correspondence between the Hamiltonian cycles of G and the stable models of R_G.

As another example, consider graph $G_2 = \langle\{1,2\}, \{\langle 1,2\rangle, \langle 2,1\rangle\}\rangle$, and interpretation

$$M' = \{\mathsf{arc}(1,2), \mathsf{arc}(2,1), \mathsf{hc}(1,2), \mathsf{hc}(2,1), \mathsf{reached}(1), \mathsf{reached}(2)\} \subseteq \mathrm{At}(R_{G_2}).$$

The reduct of R_{G_2} with respect to M' is

$$(R_{G_2})^{M'} = (R_{G_1})^M \cup \{\mathsf{arc}(1,2).\ \ \mathsf{arc}(2,1).\}.$$

It is straightforward to verify that M' is the least model of $(R_{G_2})^{M'}$. Thus, $M' \in \mathrm{SM}(R_{G_2})$, and

$$M' \cap \{\mathsf{hc}(1,1), \mathsf{hc}(1,2), \mathsf{hc}(2,1), \mathsf{hc}(2,2)\} = \{\mathsf{hc}(1,2), \mathsf{hc}(2,1)\}$$

represents a Hamiltonian cycle in G_2. ■

4 APPROACHES TO MODULARITY

Modern programming languages typically provide means to exploit *modularity* in a number of ways to govern the complexity of programs and their development process. In this chapter approaches to modularity in answer set programming and more generally in logic programming are discussed.

One essential property for a module system is *compositional semantics*. When considering how to compose a logic program from smaller parts, modules, a natural operator for composition is *union*. Classical propositional theories are *compositional* for union, and thus, given any logic programs $R_1, R_2 \in \mathcal{C}$,

$$\mathrm{CM}(R_1 \cup R_2) = \mathrm{CM}(R_1) \bowtie \mathrm{CM}(R_2), \tag{4.1}$$

where $\mathrm{CM}(R) = \{M \subseteq \mathrm{At}(R) \mid M \models R\}$ and the *natural join* $\bowtie$ of two sets of interpretations $A_1 \subseteq \mathbf{2}^{\mathrm{At}(R_1)}$ and $A_2 \subseteq \mathbf{2}^{\mathrm{At}(R_2)}$, denoted by $A_1 \bowtie A_2$, is

$$\{M_1 \cup M_2 \mid M_1 \in A_1, M_2 \in A_2, M_1 \cap \mathrm{At}(R_2) = M_2 \cap \mathrm{At}(R_1)\}. \tag{4.2}$$

The stable model semantics, however, does not lend itself directly for program composition using union. The problem is that in general, stable models associated with modules do not determine stable models assigned to their union, see [**P1,** Example 4.9], for instance.

In addition to compositional semantics, there are several properties desired from a modular logic programming language. For instance, a modular language should allow abstraction, parameterization, and information hiding and support re-usability to ease program development and maintenance of large programs [20]. In order to justify replacement of program components a non-trivial notion of program equivalence is needed. Finally, the declarativity of logic programming needs to be maintained. Modularity has been studied extensively within *conventional or Prolog-style logic programming*. In their extensive survey, Bugliesi et al.[20] identify two mainstream programming disciplines: *programming-in-the-large* and *programming-in-the-small*.

The programming-in-the-large approaches have their roots in the work of O'Keefe [103] in which logic programs are seen as an *elements of an algebra* and the operators for *composing programs* are seen as *operators in that algebra*. The fundamental idea is that a logic program should be understood as a part of a system of programs. Program composition is a powerful tool for structuring programs without any need to extend the underlying language of *Horn clauses*. Several algebraic operations such as *union*, *deletion*, *overriding union*, *closure*, *revision*, and *update* have been considered. This approach supports naturally the re-use of the pieces of programs in different composite programs, and when combined with an adequate equivalence relation also the replacement of equivalent components. Encapsulation and information hiding can be obtained by introducing suitable *interfaces* between components. For example, Mancarella and Pedreschi [90], Brogi et al. [16, 15], and Gaifman and Shapiro [48] present compositional frameworks that can be seen as different formulations of the ideas in [103].

The programming-in-the-small approaches build on ideas of Miller [95]. In his approach the composition of modules is modelled in terms of logical

connectives of a language defined as an *extension of Horn clause logic*. The approach by Giordano and Martelli [57] employs the same structural properties, but suggests a more refined way of modelling the visibility of rules than the one by Miller.

Program composition and aspects of modularity have also been considered in a number of different contexts. For instance, Etalle and Teusink [44] and Verbauten et al. [129] address compositionality issues in the context of *normal open logic programs*. Bry [18] considers compositional semantics for logic programs and *deductive databases*. Baral et al. [11] consider combinations of *knowledge bases* and introduce algorithms for composing logic programs so that satisfaction of integrity constraints is enforced. Sakama and Inoue consider *program coordination* [114, 115, 116]. In *generous coordination* (*rigorous coordination*, respectively) [116] the idea is to find a program R such that $\mathrm{SM}(R) = \mathrm{SM}(R_1) \cup \mathrm{SM}(R_2)$ ($\mathrm{SM}(R) = \mathrm{SM}(R_1) \cap \mathrm{SM}(R_2)$, respectively). In [114, 115] *maximal (minimal) consensus* is sought in terms of a program R such that its stable models are maximal (minimal) elements of the set $\{M_1 \cup M_2 \mid M_1 \in \mathrm{SM}(R_1), M_2 \in \mathrm{SM}(R_2)\}$.

Within answer set programming there is a number of approaches within involving modularity in some sense, but only few of them describe a flexible module architecture with a clearly defined interface for module interaction.

Eiter et al. [38] address modularity in the programming-in-the-small sense, and view program modules as *generalized quantifiers* [84, 97] the definitions of which are allowed to nest, that is, program R can refer to another module R' by using it as a generalized quantifier. The main program is clearly distinguished from subprograms, and it is possible to nest calls to submodules if the so-called *call graph* is *hierarchical*, that is, *acyclic*. Nesting, however, raises the computational complexity depending on the depth of nesting. Ianni et al. [60] have another programming-in-the-small approach based on *templates*. The semantics of programs containing template atoms is determined by an *explosion algorithm*, which basically replaces the template with a standard logic program. However, the explosion algorithm is not guaranteed to terminate if template definitions are used recursively. Baral et al. [9] use *macros*, but again, acyclic dependencies between modules are assumed. Tari et al. [123] extend the language of normal logic programs by introducing the concept of *import rules* for their ASP program modules. There are three types of import rules which are used to import a set of tuples $\overline{X}$ for a predicate q from another module. An *ASP module* is defined as a quadruple of a module name, a set of parameters, a collection of normal rules and a collection of import rules. Semantics is only defined for modular programs with acyclic dependency graph, and answer sets of a module are defined with respect to the modular ASP program containing it. Also, it is required that import rules referring to the same module always have the same form.

Programming-in-the-large approaches to modularity in ASP are mostly based on Lifschitz and Turner's splitting set theorem [80] or are variants of it. A component structure induced by a *splitting sequence*, that is, iterated splittings of a program, allows a bottom-up computation of answer sets. The restriction induced is that the dependency graph of the component chain needs to be acyclic. Eiter et al. [37] consider disjunctive logic programs as a query language for relational databases. A query program π is instantiated

with respect to an input database D confined by an input schema $\mathbf{R}$. The semantics of π determines, for example, the answer sets of $\pi[D]$ which are projected with respect to an output schema $\mathbf{S}$. Module architecture is based on both *positive and negative dependencies* and no recursion between modules is tolerated. These constraints enable a straightforward generalization of the splitting set theorem for the architecture. Faber et al. [45] consider *independent sets* which are a specialization of splitting sets. In [51, 7] an approach based on *lp-functions* is proposed. An lp-function has an interface based on input and output signatures. Several operations, for instance *incremental extension*, *interpolation*, *input opening*, and *input extension*, are introduced for composing and refining lp-functions. The composition of lp-functions, however, only allows incremental extension, and thus similarly to the splitting set theorem there can be no recursion between lp-functions.

The remaining of this chapter is dedicated to a more detailed description of the module framework by Gaifman and Shapiro [48], and the splitting sets by Lifschitz and Turner [80]. These approaches are then contrasted with a module architecture proposed for answer set programming in [**P1, P2**].

4.1 MODULE FRAMEWORK BY GAIFMAN AND SHAPIRO

The language considered in [48] is that of *definite logic programs*, that is, clauses of the form $A \leftarrow B_1, \ldots, B_n$, where A, B_1,..., B_n are atoms. Atoms are predicates instantiated with *terms*, and can thus contain *function symbols* and *variables* in addition to constants. The semantics is based on *atomic consequences*, that is, an atom A is a logical consequence of a program P if and only if A is derivable from R via *SLD resolutions*. This is different from answer set programming where the semantics is based on models. However, the minimal models considered in ASP coincide with atomic consequences for *propositional positive logic programs*.

A *logic program module* $\Pi = \langle R, Im, Ex, Int \rangle$ is a set of clauses R with partitioning of predicates into *imported*, *exported* and *internal* ones. An imported predicate is supplied to the module by the environment, for example, another module, and it cannot appear in the head of a clause. Other predicates can appear anywhere. External predicates can be supplied to other modules, while internal predicates cannot. Communication between modules is achieved through predicate sharing. Two logic program modules $\Pi_1 = \langle R_1, Im_1, Ex_1, Int_1 \rangle$ and $\Pi_2 = \langle R_2, Im_2, Ex_2, Int_2 \rangle$ are *composable*, if Int_1 and Int_2 are local to Π_1 and Π_2 and $Ex_1 \cap Ex_2 = \emptyset$. Composition of modules is defined as

$$\Pi_1 \oplus \Pi_2 = \langle R_1 \cup R_2, (Im_1 \cup Im_2) \setminus (Ex_1 \cup Ex_2), \\ Ex_1 \cup Ex_2, Int_1 \cup Int_2 \rangle, \quad (4.3)$$

that is, basically by taking the union of the sets of clauses and by updating the module interface accordingly.

Semantics for modules is defined by taking into account the interface restrictions. A clause $A \leftarrow B_1, \ldots, B_n$ is an *Import/Export clause* (I/E-clause) of Π, if $A \in Ex$ and $\{B_1, \ldots, B_n\} \subseteq Im$. An *I/E consequence* of Π is a

logical consequence of Π which is an I/E clause, and an *atomic I/E consequence* is an *atomic* consequence whose predicate is exported. Furthermore, modules Π_1 and Π_2 are semantically equivalent if and only if Π_1 and Π_2 have the same *minimal* [1] *I/E consequences*, which coincide with the atomic I/E consequences [48, Theorem 11].

4.2 MODULE STRUCTURE BASED ON SPLITTING SETS

The splitting set theorem is formulated here for programs in $\mathcal{C}_{\mathrm{d}}$ instead of the class of *extended disjunctive logic programs* in [80].

Definition 4.1. *A splitting set for $R \in \mathcal{C}_{\mathrm{d}}$ is any set $U \subseteq \mathrm{At}(R)$ such that for every rule $r \in R$, if $\mathsf{head}(r) \cap U \neq \emptyset$, then $\mathsf{head}(r) \cup \mathsf{body}(r) \subseteq U$.*

The set of rules $r \in R$ such that $\mathsf{head}(r) \cup \mathsf{body}(r) \subseteq U$ is the *bottom* of R relative to U, denoted by $\mathrm{b}_U(R)$. The set $\mathrm{t}_U(R) = R \setminus \mathrm{b}_U(R)$ is the *top* of R relative to U which can be partially evaluated with respect to an interpretation $X \subseteq U$. The result is a program $\mathrm{e}(\mathrm{t}_U(R), X)$ which contains a rule

$$a_1 \vee \ldots \vee a_h \leftarrow b'_1 \ldots, b'_k, {\sim}c'_1, \ldots, {\sim}c'_l$$

for each rule (3.9) in $\mathrm{t}_U(R)$ such that

- $\{b'_1 \ldots, b'_k\} = \{b_1, \ldots, b_n\} \setminus U$ and $\{c'_1 \ldots, c'_l\} = \{c_1, \ldots, c_m\} \setminus U$,
- $\{b_1, \ldots, b_n\} \cap U \subseteq X$, and
- $(\{c_1, \ldots, c_m\} \cap U) \cap X = \emptyset$.

A *solution* to a program $R \in \mathcal{C}_{\mathrm{d}}$ with respect to a splitting set U is a pair $\langle X, Y \rangle$ such that

(i) $X \subseteq U$ is a stable model of $\mathrm{b}_U(R)$, and

(ii) $Y \subseteq \mathrm{At}(R) \setminus U$ is a stable model of $\mathrm{e}(\mathrm{t}_U(R), X)$.

Solutions and stable models relate as follows.

Theorem 4.2 (The splitting set theorem [80])**.** *For any splitting set U for $R \in \mathcal{C}_{\mathrm{d}}$ and $M \subseteq \mathrm{At}(R)$, it holds $M \in \mathrm{SM}(R)$ if and only if $\langle M \cap U, M \setminus U \rangle$ is a solution to R with respect to U.*

In particular, one may notice that there is no *recursion* between the bottom and the top part of a program, and in this case the stable model semantics becomes compositional.

The splitting set theorem can also be used in an iterative manner, if there is a *monotone sequence* of splitting sets $\langle U_1, \ldots, U_i, \ldots \rangle$, that is, $U_i \subset U_j$ if $i < j$, for program $R \in \mathcal{C}_{\mathrm{d}}$. This is called a *splitting sequence* and it induces a *component structure* for R. The splitting set theorem generalizes to a

[1] A clause C is minimal in V if it is not tautological, there is no proper subclause of C in V, and C is not an instance of another $C' \in V$ with the same number of literals. An I/E consequence is minimal if it is minimal in the set of I/E consequences.

splitting sequence theorem [80], and given a splitting sequence, stable models of program R can be computed iteratively bottom-up. The component structure based on a splitting sequence does not have a specified interface for communication between the components but the incremental structure can be used as a basis for input/output relation for the parts.

4.3 ACHIEVING THE COMPOSITIONALITY OF STABLE MODEL SEMANTICS

In [**P1**] a module architecture inspired by the approach by Gaifman and Shapiro [48] is proposed for SMODELS programs under the stable model semantics. In [**P2**] a similar module system is introduced for the class of disjunctive logic programs. Moreover, the framework in [**P1, P2**] generalizes the component structure based on a splitting sequence [80] by showing that compositionality of the stable model semantics can be achieved even if negative recursion between modules is allowed.

A logic program module[2] is a quadruple in analogy to [48].

Definition 4.3. *A logic program module* Π *over* $\mathcal{C}$ *is* $\langle R, I, O, H\rangle$ *where*

1. $R \in \mathcal{C}$ *is a finite set of rules;*
2. I*,* O*, and* H *are pairwise disjoint sets of input, output, and hidden atoms;*
3. $\mathrm{At}(R) \subseteq \mathrm{At}(\Pi)$ *defined by* $\mathrm{At}(\Pi) = I \cup O \cup H$*; and*
4. $\mathsf{head}(R) \cap I = \emptyset$.

In the following $\mathcal{M}_\mathrm{n}$, $\mathcal{M}_\mathrm{s}$, $\mathcal{M}_\mathrm{d}$, and $\mathcal{M}$ are used to denote the sets of all modules over $\mathcal{C}_\mathrm{n}$, $\mathcal{C}_\mathrm{s}$, $\mathcal{C}_\mathrm{d}$, and $\mathcal{C}$, respectively. The atoms in $\mathrm{At}_\mathrm{v}(\Pi) = I \cup O$ are considered to be *visible* and hence accessible to other modules conjoined with Π either to produce input for Π or to utilize the output of Π. The *hidden* atoms in $\mathrm{At}_\mathrm{h}(\Pi) = H = \mathrm{At}(\Pi) \setminus \mathrm{At}_\mathrm{v}(\Pi)$ are used to formalize some auxiliary concepts of Π which may not be sensible for other modules but may save space substantially, see [**P3**, Example 4.5], for instance. The condition $\mathsf{head}(R) \cap I = \emptyset$ ensures that a module may not interfere with its own input by defining input atoms of I in terms of its rules. Thus, input atoms are only allowed to appear as conditions in rule bodies.

The stable model semantics is generalized to cover modules by introducing a generalization of the Gelfond-Lifschitz reduct [**P1, P2**]. In addition to negative literals also *literals involving input atoms* get evaluated in the reduction.

Definition 4.4. *Given a logic program module* $\Pi = \langle R, I, O, H\rangle \in \mathcal{M}$*, the* reduct *of* R *under input signature* I *with respect to an interpretation* $M \subseteq \mathrm{At}(\Pi)$*, denoted by* R^M_I*, contains*

[2] In [**P2**], instead of referring to modules, a concept of a DLP-function is used in analogy to lp-functions [51].

- *a rule* $a \leftarrow b'_1, \ldots, b'_k$ *if and only if there is a choice rule (3.4) in* R *such that* $a \in \{a_1, \ldots, a_h\} \cap M$ *and*

$$\{b'_1, \ldots, b'_k\} = \{b_1, \ldots, b_n\} \setminus I, \tag{4.4}$$
$$\{b_1, \ldots, b_n\} \cap I \subseteq M, \text{ and} \tag{4.5}$$
$$M \cap \{c_1, \ldots, c_m\} = \emptyset; \tag{4.6}$$

- *a rule* $a \leftarrow w' \leq \{b'_1 = w_{b'_1}, \ldots, b'_k = w_{b'_k}\}$ *if and only if there is a weight rule (3.3) in* R *such that* $\{b'_1, \ldots, b'_k\} = \{b_1, \ldots, b_n\} \setminus I$ *and*

$$w' = \max(0, w - \sum_{b \in \{b_1, \ldots, b_m\} \cap I \cap M} w_b - \sum_{c \in \{c_1, \ldots, c_m\} \setminus M} w_c); \text{ and}$$

- *a rule* $a_1 \vee \ldots \vee a_h \leftarrow b'_1, \ldots, b'_k$ *if and only if there is a disjunctive rule (3.9) in* R *such that conditions (4.4)–(4.6) hold.*

The generalized reduct R_I^M is a positive logic program and $\mathrm{At}(R_I^M) \subseteq \mathrm{At}(R) \setminus I$. Thus, the stable model semantics can directly be generalized for logic program modules.

Definition 4.5. *An interpretation* $M \subseteq \mathrm{At}(\Pi)$ *is a stable model of a module* $\Pi = \langle R, I, O, H \rangle \in \mathcal{M}$*, denoted by* $M \in \mathrm{SM}(\Pi)$*, if and only if* $M \setminus I \in \mathrm{MM}(R_I^M)$.

There are also alternative ways to handle input atoms. One possibility is to combine a module with a set of facts (or a database) over its input signature [102, 100]. Yet another approach is to interpret input atoms as *fixed atoms* in the sense of parallel circumscription [74] as done in [**P7**]. Furthermore, the alternative definition of stable models for logic programs based on the *classical models* of the *completion* of a program [24] and its *loop formulas* [82], is extended to cover $\mathcal{M}_{\mathrm{n}}$ in [**P1**].

Following [48] it is initially assumed that modules $\Pi_1 = \langle R_1, I_1, O_1, H_1 \rangle \in \mathcal{M}$ and $\Pi_2 = \langle R_2, I_2, O_2, H_2 \rangle \in \mathcal{M}$, may be put together when their output signatures are disjoint, that is, $O_1 \cap O_2 = \emptyset$, and they *respect each other's hidden atoms*, that is, $H_1 \cap \mathrm{At}(\Pi_2) = \emptyset$ and $H_2 \cap \mathrm{At}(\Pi_1) = \emptyset$. Then their *composition* is

$$\Pi_1 \oplus \Pi_2 = \langle R_1 \cup R_2, (I_1 \cup I_2) \setminus (O_1 \cup O_2), O_1 \cup O_2, H_1 \cup H_2 \rangle$$

in analogy to (4.3). However, the conditions given for $\oplus$ are not enough to guarantee compositionality in the case of stable models. For example, in [**P1,** Example 4.9], modules $\Pi_1 = \langle \{a \leftarrow b.\}, \{b\}, \{a\}, \emptyset \rangle$ and $\Pi_2 = \langle \{b \leftarrow a.\}, \{a\}, \{b\}, \emptyset \rangle$ are considered. They both have stable models $\emptyset$ and $\{a, b\}$ by symmetry, but $\{a, b\}$ is not a stable model of their composition $\Pi_1 \oplus \Pi_2 = \langle \{a \leftarrow b.\ b \leftarrow a.\}, \emptyset, \{a, b\}, \emptyset \rangle$.

By assuming additionally that there is no recursion between the modules, the splitting set theorem [80] allows for computing the stable models bottom-up, that is, a splitting of a program can be used as a basis for a module structure as follows. If U is a splitting set for a program $R \in \mathcal{C}_{\mathrm{d}}$, then R can be defined as composition

$$\Pi_B \oplus \Pi_T = \langle \mathrm{b}_U(R), \emptyset, U, \emptyset \rangle \oplus \langle \mathrm{t}_U(R), U, \mathrm{At}(R) \setminus U, \emptyset \rangle.$$

Now, by Theorem 4.2, $M \in \mathrm{SM}(R)$ if and only if $\langle M \cap U, M \setminus U \rangle$ is a solution for R with respect to U. This implies $M \cap U \in \mathrm{SM}(\Pi_B)$ and $M \in \mathrm{SM}(\Pi_T)$, and furthermore, $M \in \mathrm{SM}(\Pi_B) \bowtie \mathrm{SM}(\Pi_T)$.

In [**P1, P2**] it is shown that it is possible to go beyond the splitting set theorem, as only positive recursion between modules turns out to be problematic. Formally, the positive dependency graph of a module $\Pi = \langle R, I, O, H \rangle \in \mathcal{M}$ is defined as $\mathrm{Dep}^+(\Pi) = \mathrm{Dep}^+(R)$[3]. Given that $\Pi_1 \oplus \Pi_2$ is defined, $\Pi_1 = \langle R_1, I_1, O_1, H_1 \rangle$ and $\Pi_2 = \langle R_2, I_2, O_2, H_2 \rangle$ are *mutually dependent* if and only if $\mathrm{Dep}^+(\Pi_1 \oplus \Pi_2)$ has an strongly connected component S such that $S \cap O_1 \neq \emptyset$ and $S \cap O_2 \neq \emptyset$, that is, the SCC S is shared by Π_1 and Π_2.

Definition 4.6. *Given modules $\Pi_1, \Pi_2 \in \mathcal{M}$, their join is*

$$\Pi_1 \sqcup \Pi_2 = \Pi_1 \oplus \Pi_2$$

provided that (i) $\Pi_1 \oplus \Pi_2$ is defined and (ii) Π_1 and Π_2 are mutually independent.

The conditions in Definition 4.6 impose no restrictions on positive dependencies *inside* modules or on *negative* dependencies in general. Moreover, if modules Π_1 and Π_2 are mutually independent, then the condition $O_1 \cap O_2 = \emptyset$ for their composition $\Pi_1 \oplus \Pi_2$ holds automatically. Thus, the requirement of mutual independence can also be viewed as a variant of condition $O_1 \cap O_2 = \emptyset$ restoring the compositionality of the module system under the stable model semantics.

As modules may involve hidden atoms, (4.2) needs to be accommodated to the modular context. Given modules Π_1 and Π_2 such that $\Pi_1 \oplus \Pi_2$ is defined, the natural join of $A_1 \subseteq \mathbf{2}^{\mathrm{At}(\Pi_1)}$ and $A_2 \subseteq \mathbf{2}^{\mathrm{At}(\Pi_2)}$ is

$$A_1 \bowtie A_2 = \{M_1 \cup M_2 \mid M_1 \in A_1, M_2 \in A_2, M_1 \text{ and } M_2 \text{ are compatible}\},$$

where $M_1 \subseteq \mathrm{At}(\Pi_1)$ and $M_2 \subseteq \mathrm{At}(\Pi_2)$ are compatible if and only if $M_1 \cap \mathrm{At_v}(\Pi_2) = M_2 \cap \mathrm{At_v}(\Pi_1)$.

If a program (module) consists of several submodules, its stable models are locally stable for the respective submodules; and on the other hand, local stability implies global stability for compatible stable models of the submodules, that is, the stable model semantics of modules is compositional for $\sqcup$.

Theorem 4.7 (The module theorem)**.** *If $\Pi_1, \Pi_2 \in \mathcal{M}$ are modules such that $\Pi_1 \sqcup \Pi_2$ is defined, then* $\mathrm{SM}(\Pi_1 \sqcup \Pi_2) = \mathrm{SM}(\Pi_1) \bowtie \mathrm{SM}(\Pi_2)$.

The module theorem also straight-forwardly generalizes for a collection of modules [**P1, P2**].

Example 4.8. Recall the encoding for solving Hamiltonian cycle problem for a directed graph $G = \langle N, E \rangle$ given in Example 3.1. The encoding is general in the sense that only the set of facts (3.10) is dependent of the graph G, and by chancing the facts, the encoding works for any directed graph of n nodes.

[3] Since the definitions of input atoms are external to a module $\Pi = \langle R, I, O, H \rangle \in \mathcal{M}$, the input atoms could be excluded from the positive dependency graph of Π, that is, $\mathrm{Dep}^+(\Pi)$ can be defined as the graph with $O \cup H$ and $\{\langle b, a \rangle \in (O \cup H) \times (O \cup H) \mid a \leq_1 b\}$ as the sets of nodes and edges, respectively.

Looking this from the point of view of the module architecture, a particular graph G can be viewed as an input for the encoding. With respect to the solutions of the problem, the atoms describing the Hamiltonian cycles are particularly interesting in the stable models of the encoding. Other atoms can be viewed as internal information. Thus, one may define $\Pi^n_{\mathrm{HC}} = \langle R, I, O, H\rangle$, where the set of rules R is the union of the sets in (3.11)–(3.19),

$$\begin{aligned} I &= \{\mathsf{arc}(x,y) \mid 1 \leq x, y \leq n\}, \\ O &= \{\mathsf{hc}(x,y) \mid 1 \leq x, y \leq n\}, \text{ and} \\ H &= \{\mathsf{reached}(x) \mid 1 \leq x \leq n\} \cup \{c, d, e\}. \end{aligned}$$

Now, module Π^n_{HC} solves the Hamiltonian cycle problem for all directed graphs of n nodes.

Furthermore, one may notice that the problem can be partitioned into two subproblems, that is, selecting edges to a cycle candidate and checking that each node is reachable along the edges in the candidate. The subtasks can be separated in different modules. A module $\Pi^n_{\mathrm{H}} = \langle R_{\mathrm{H}}, I, O, \{c, d\}\rangle$ selects the edges to be taken into a cycle. The set of rules R_{H} is the union of the sets in (3.11)–(3.16). Another module $\Pi^n_{\mathrm{R}} = \langle R_{\mathrm{R}}, I', \emptyset, H'\rangle$ performs the reachability check. The input signature of Π^n_{R} is $I' = O$, and all other atoms are hidden $H' = \{\mathsf{reached}(x) \mid 1 \leq x \leq n\} \cup \{e\}$. The set of rules R_{R} is the union of the sets in (3.17)–(3.19). Modules Π^n_{H} and Π^n_{R} respect each other's hidden atoms and are mutually independent. Thus, their join $\Pi^n_{\mathrm{H}} \sqcup \Pi^n_{\mathrm{R}} = \langle R_{\mathrm{H}} \cup R_{\mathrm{R}}, I, O, H\rangle$ is defined, and moreover $\Pi^n_{\mathrm{H}} \sqcup \Pi^n_{\mathrm{R}} = \Pi^n_{\mathrm{HC}}$.

As an illustration of the module theorem, consider $n = 2$, and the interpretation $M = \{\mathsf{hc}(1,2), \mathsf{hc}(2,1), \mathsf{reached}(1), \mathsf{reached}(2)\}$ used in Examples 3.3 and 3.5. Now, $M \cap \mathrm{At}(\Pi^2_{\mathrm{R}})$ is a stable model of Π^2_{R}, since given such a selection of edges to a cycle, both nodes are reachable. However, $N = M \cap \mathrm{At}(\Pi^2_{\mathrm{H}}) = \{\mathsf{hc}(1,2), \mathsf{hc}(2,1)\}$ is not a stable model of Π^2_{H}, since $(R_{\mathrm{H}})^N_I$ contains the rules

$$\begin{array}{ll} c \leftarrow 2 \leq \{\mathsf{hc}(1,1), \mathsf{hc}(1,2)\}. & c \leftarrow 2 \leq \{\mathsf{hc}(2,1), \mathsf{hc}(2,2)\}. \\ c \leftarrow 2 \leq \{\mathsf{hc}(1,1), \mathsf{hc}(2,1)\}. & c \leftarrow 2 \leq \{\mathsf{hc}(1,2), \mathsf{hc}(2,2)\}. \\ d \leftarrow c. & \end{array}$$

and $\mathrm{LM}((R_{\mathrm{H}})^N_I) = \emptyset$. By the module theorem, it follows that M is not a stable model of the join $\Pi^2_{\mathrm{H}} \sqcup \Pi^2_{\mathrm{R}} = \Pi^2_{\mathrm{HC}}$. ■

The module theorem strengthens an earlier version given in [64] to cover programs that involve positive body literals. Moreover, the module theorem is a proper generalization of the splitting set theorem [80], see [**P1,** Example 4.31] for an illustration. The main difference is that splitting sets do not enable any kind of recursion between modules. The module theorem can be used in a very similar fashion to the splitting set theorem, see [**P1,** Examples 4.17 and 4.18]. The module theorem demonstrates the feasibility of the respective module architecture, but it is also applied similarly to the splitting set theorem as a tool to simplify mathematical proofs in [**P1, P2, P4, P7**].

Furthermore, in [**P1**] a technique for obtaining a *module-decomposition* for a complete program $R \in \mathcal{C}_{\mathrm{s}}$ is proposed, based on the strongly connected components of $\mathrm{Dep}^+(R)$ and visibility information of $\mathrm{At}(R) = \mathrm{At}_{\mathrm{v}}(R) \cup$

$\mathrm{At_h}(R)$. Similar technique can be applied to programs in $\mathcal{C}_\mathrm{d}$. Notice, however, that disjunctive rules may not be projected in the same way as choice rules. Thus, one has to make sure that all the head atoms sharing a rule are placed within one component. A more detailed knowledge of the internal structure of a program might reveal ways to improve search for stable models. Another application can be found in modularization of the translation-based equivalence verification method to be discussed in Section 5.2.

There are cases in which $\Pi_1 \sqcup \Pi_2$ is not defined while $\mathrm{SM}(\Pi_1 \oplus \Pi_2) = \mathrm{SM}(\Pi_1) \bowtie \mathrm{SM}(\Pi_2)$ holds, see [**P1,** Example 5.4], for instance. This suggests that even the denial of positive recursion between modules can be relaxed in certain cases.

Definition 4.9. *Given two modules* $\Pi_1, \Pi_2 \in \mathcal{M}$*, their semantical join is* $\Pi_1 \underline{\sqcup} \Pi_2 = \Pi_1 \oplus \Pi_2$ *provided that (i)* $\Pi_1 \oplus \Pi_2$ *is defined and (ii)* $\mathrm{SM}(\Pi_1 \oplus \Pi_2) = \mathrm{SM}(\Pi_1) \bowtie \mathrm{SM}(\Pi_2)$.

The module theorem holds by definition for modules composed with $\underline{\sqcup}$. The tradeoff between the syntactical and the semantical condition lies in the computational complexity of checking whether the condition for the respective composition is satisfied. The syntactical restriction denying positive recursion between modules is easy to check, since SCCs can be found in a linear time with respect to the size of the dependency graph [124]. To the contrary, checking whether $\mathrm{SM}(\Pi_1 \oplus \Pi_2) = \mathrm{SM}(\Pi_1) \bowtie \mathrm{SM}(\Pi_2)$ is a computationally harder problem. For instance, for modules in $\mathcal{M}_\mathrm{s}$ it forms a **coNP**-complete decision problem [**P1**], and, the result holds even for modules in $\mathcal{M}_\mathrm{n} \subset \mathcal{M}_\mathrm{s}$.

Earlier approaches to modularity within ASP [38, 60, 123, 80, 37, 45, 51, 7] typically assume acyclic dependency graph between modules. The module system in [**P1, P2**] allows a more flexible way of combining modules. The restrictions imposed on the interface of an individual module, and the requirement of mutual independence of modules joined together can be regarded as part of good programming style in answer set programming. The characteristics of ASP modules proposed in [**P1, P2**] include a clearly defined interface for interaction between other modules in analogy to [48]. Moreover, to justify a replacement of a module with another within a larger program, an equivalence relation, namely *modular equivalence* is proposed [**P1, P2**]. Properties of modular equivalence among with other equivalence relations proposed for logic programs are discussed in the next chapter.

5 EQUIVALENCE RELATIONS

There are several notions of equivalence proposed for logic programs under the stable model semantics. A summary of equivalence relations for logic programs in terms the congruence property and the preservation of the number of stable models is given in Table 5.1.

Lifschitz et al. [78] address the notions of *weak/ordinary equivalence* and *strong equivalence*. Programs $R_1, R_2 \in \mathcal{C}$ are weakly equivalent, denoted by $R_1 \equiv R_2$, if and only if $\mathrm{SM}(R_1) = \mathrm{SM}(R_2)$; and strongly equivalent, denoted by $R_1 \equiv_\mathrm{s} R_2$, if and only if $R_1 \cup R \equiv R_2 \cup R$ for each program $R \in \mathcal{C}$. The program R in the above definition can be understood as an arbitrary context in which the two programs being compared could be placed. Therefore strongly equivalent logic programs are semantics preserving substitutes of each other and relation $\equiv_\mathrm{s}$ is a *congruence relation* for $\cup$ among programs in $\mathcal{C}$, that is, if $R_1 \equiv_\mathrm{s} R_2$, then also $R_1 \cup R \equiv_\mathrm{s} R_2 \cup R$ for all $R \in \mathcal{C}$.

A drawback of the relation $\equiv_\mathrm{s}$ is that it is quite restrictive, allowing only rather straightforward semantics-preserving transformations of (sets of) rules, see [78, 105, 81, 126, 21] for characterizations of strong equivalence. Lifschitz et al. [78] characterize $\equiv_\mathrm{s}$ in Heyting's logic *here-and-there* which is an intermediary logic between intuitionistic and classical propositional logics. This result implies that each program transformation admitted by $\equiv_\mathrm{s}$ is based on a classical equivalence of the part being replaced R_1 and its substitute R_2, that is, $R_1 \equiv_\mathrm{s} R_2$ implies that R_1 and R_2 are classically equivalent (R_1 and R_2 have the same classical models). However, the converse is not true in general as there are classically equivalent programs that are not strongly equivalent. In the other extreme, weak equivalence relates programs that admit exactly the same behavior under the stable model semantics. However, weak equivalence is not a congruence for the union of programs, which limits its usefulness, especially in a modular case.

A way to weaken strong equivalence is to restrict possible contexts to sets of facts. The notion of *uniform equivalence* has its roots in the database community [111, 87], see [33] for the context of answer set programming. Programs $R_1, R_2 \in \mathcal{C}$ are uniformly equivalent, denoted by $R_1 \equiv_\mathrm{u} R_2$, if and only if $R_1 \cup F \equiv R_2 \cup F$ for any set F of facts. Eiter et al. [34, Example 1] show that $R_1 \equiv_\mathrm{u} R_2$ does not imply $R_1 \equiv_\mathrm{s} R_2$ in general. This implies that, similarly to $\equiv$, uniform equivalence fails to be a congruence for union of programs.

There are also *relativized variants of strong and uniform equivalence* [81, 133] which allow the context R to be constrained by a set of atoms A, that is, only contexts R such that $\mathrm{At}(R) \subseteq A$ are considered. Inoue and Sakama propose a variant of relativized equivalence based on *update equivalence* [62]. Programs $R_1, R_2 \in \mathcal{C}$ are *strongly equivalent with respect to a set* $\mathcal{R} \in \mathcal{C}$ *of rules* if and only if $R_1 \cup R \equiv R_2 \cup R$ for any set $R \subseteq \mathcal{R}$.

Woltran recently presented a general framework characterizing $\langle \mathcal{H}, \mathcal{B} \rangle$-*equivalence* [134]. The definition of $\langle \mathcal{H}, \mathcal{B} \rangle$-equivalence is similar to that of strong equivalence, but the set of possible contexts is restricted by limiting the head and body occurrences of atoms in a context program R by $\mathcal{H}$ and $\mathcal{B}$, respectively. Thus, programs R_1 and R_2 are $\langle \mathcal{H}, \mathcal{B} \rangle$-equivalent if and only if

$R_1 \cup R \equiv R_2 \cup R$ for all R such that $\mathsf{head}(R) \subseteq \mathcal{H}$ and $\mathsf{body}(R) \subseteq \mathcal{B}$. Several notions of equivalence such as weak equivalence together with (relativized) strong and (relativized) uniform equivalence can be seen as special cases of $\langle \mathcal{H}, \mathcal{B} \rangle$-equivalence by varying the sets $\mathcal{H}$ and $\mathcal{B}$.

All the equivalence relations introduced here so far basically assume that the set of atoms appearing in the programs R_1 and R_2 under consideration are the same. This makes these relations less useful if $\mathrm{At}(R_1)$ and $\mathrm{At}(R_2)$ differ by some local atoms not trivially false in all stable models, see [**P3,** Example 4.5], for instance. To solve this problem, the atoms in $\mathrm{At}(R)$ may be partitioned into $\mathrm{At_v}(R)$ and $\mathrm{At_h}(R)$ to determine the *visible* and the *hidden* parts of $\mathrm{At}(R)$ in analogy to a module interface. The idea is that visible atoms form an interface for interaction between programs, and hidden atoms are local to each program and thus negligible when equivalence of programs is concerned. A very general framework based on *equivalence frames* [40] captures various kinds of equivalence relations. Equivalence frames based on *projected answer sets* enable ignoring the hidden atoms when equivalence of programs is of interest. However, a *projective equivalence*, defined as $R_1 \equiv_\mathrm{p} R_2$ if and only if

$$\{M \cap \mathrm{At_v}(R_1) \mid M \in \mathrm{SM}(R_1)\} = \{N \cap \mathrm{At_v}(R_2) \mid N \in \mathrm{SM}(R_2)\}, \quad (5.1)$$

may not preserve the number of stable models. This is somewhat unsatisfactory because of the general nature of answer set programming. The stable models of a program typically correspond to the solutions of the problem being solved and thus the exact preservation of models is highly significant.

The *visible equivalence relation* [64] strives for a strict correspondence. Programs $R_1, R_2 \in \mathcal{C}$ are visibly equivalent, denoted by $R_1 \equiv_\mathrm{v} R_2$, if and only if $\mathrm{At_v}(R_1) = \mathrm{At_v}(R_2)$ and there is a bijection $f : \mathrm{SM}(R_1) \to \mathrm{SM}(R_2)$ such that for all $M \in \mathrm{SM}(R_1)$, $M \cap \mathrm{At_v}(R_1) = f(M) \cap \mathrm{At_v}(R_2)$. The number of stable models is preserved under $\equiv_\mathrm{v}$, and in the absence of hidden atoms, the relation $\equiv_\mathrm{v}$ becomes very close to $\equiv$. The only difference is the requirement $\mathrm{At}(R_1) = \mathrm{At}(R_2)$ insisted by $\equiv_\mathrm{v}$. Thus, visible equivalence is not a congruence for program union either, which limits its usefulness in a modular setting.

Table 5.1: Properties of equivalence relations for logic programs.

	congruence for $\cup$	preserves number of stable models
$\equiv$	no	yes
$\equiv_\mathrm{u}$	no	yes
$\equiv_\mathrm{s}$	yes	yes
$\equiv_\mathrm{p}$	no	no
$\equiv_\mathrm{v}$	no	yes

5.1 MODULE-LEVEL EQUIVALENCE

The module system proposed in [**P1, P2**] enables lifting equivalence relations to the level of modules. For instance, program modules $\Pi_1, \Pi_2 \in \mathcal{M}$ are visibly equivalent, denoted by $\Pi_1 \equiv_{\mathrm{v}} \Pi_2$ if and only if $\mathrm{At}_{\mathrm{v}}(\Pi_1) = \mathrm{At}_{\mathrm{v}}(\Pi_2)$ and there is a bijection $f : \mathrm{SM}(\Pi_1) \to \mathrm{SM}(\Pi_2)$ such that for all $M \in \mathrm{SM}(\Pi_1)$, $M \cap \mathrm{At}_{\mathrm{v}}(\Pi_1) = f(M) \cap \mathrm{At}_{\mathrm{v}}(\Pi_2)$. Taking into account the input/output interface of modules, a more refined relation of *modular equivalence* is obtained based on visible equivalence. Program modules $\Pi_1 = \langle R_1, I_1, O_1, H_1 \rangle, \Pi_2 = \langle R_2, I_2, O_2, H_2 \rangle \in \mathcal{M}$ are modularly equivalent, denoted by $\Pi_1 \equiv_{\mathrm{m}} \Pi_2$ if and only if $I_1 = I_2$ and $\Pi_1 \equiv_{\mathrm{v}} \Pi_2$.

Modular equivalence lends itself for program substitutions in analogy to strong equivalence [78] when considering the join operator $\sqcup$, that is, relation $\equiv_{\mathrm{m}}$ is a proper *congruence* for join [**P1, P2**].

Theorem 5.1 (Congruence). *Let $\Pi_1, \Pi_2, \Pi \in \mathcal{M}$ be modules such that the joins $\Pi_1 \sqcup \Pi$ and $\Pi_2 \sqcup \Pi$ are defined. If $\Pi_1 \equiv_{\mathrm{m}} \Pi_2$, then $\Pi_1 \sqcup \Pi \equiv_{\mathrm{m}} \Pi_2 \sqcup \Pi$.*

Thus, modular equivalence can be viewed as a reasonable compromise between uniform equivalence [33] and strong equivalence [78]. Uniform equivalence fails to be a congruence for union, whereas strong equivalence is a congruence. Strong equivalence, however, allows only rather straightforward semantics-preserving transformations of sets of rules, whereas uniform equivalence is less restrictive.

In [**P1,** Example 4.21] it is illustrated how the join effectively prunes contexts, which would alter the semantics of input atoms. The congruence property also holds for the semantical join $\sqcup$ by definition.

Example 5.2. Module $\Pi^n_{\mathrm{HR}} = \langle R'', I, O, H'' \rangle$ is based on an alternative encoding for Hamiltonian cycle problem given in [118]. In contrast to the encoding described in Example 4.8, this encoding does not allow to separate the selection of the edges to the cycle and the checking of reached nodes into separate modules as their definitions are mutually dependent. The input signature of Π^n_{HR} is the same as for Π^n_{H}, that is, $I = \{\mathsf{arc}(x, y) \mid 1 \leq x, y \leq n\}$. The output signature of Π^n_{HR} is the output signature of $\Pi^n_{\mathrm{H}} \sqcup \Pi^n_{\mathrm{R}}$, that is, $O = \{\mathsf{hc}(x, y) \mid 1 \leq x, y \leq n\}$, and the rest of the atoms are hidden, that is,

$$H'' = \{\mathsf{reached}(x) \mid 1 \leq x \leq n\} \cup \{f\}.$$

The set of rules R'' contains the following rules:

$$\begin{aligned}
\{\mathsf{hc}(1, x)\} &\leftarrow \mathsf{arc}(1, x). \\
\{\mathsf{hc}(x, y)\} &\leftarrow \mathsf{reached}(x), \mathsf{arc}(x, y). \\
\mathsf{reached}(y) &\leftarrow \mathsf{hc}(x, y). \\
f &\leftarrow \sim f, \sim \mathsf{reached}(x). \\
f &\leftarrow \sim f, \mathsf{hc}(x, y), \mathsf{hc}(x, z). \qquad (5.2) \\
f &\leftarrow \sim f, \mathsf{hc}(x, y), \mathsf{hc}(z, y). \qquad (5.3)
\end{aligned}$$

for each $1 \leq x, y, z \leq n$ such that $y \neq z$ in (5.2) and $x \neq z$ in (5.3).

Now, Π^n_{HR} and $\Pi^n_{\mathrm{H}} \sqcup \Pi^n_{\mathrm{R}}$ have the same input/output interface, and one may check that $\mathrm{SM}(\Pi^n_{\mathrm{HR}}) = \mathrm{SM}(\Pi^n_{\mathrm{H}} \sqcup \Pi^n_{\mathrm{R}})$. This implies $\Pi^n_{\mathrm{HR}} \equiv_{\mathrm{m}} \Pi^n_{\mathrm{H}} \sqcup \Pi^n_{\mathrm{R}}$.

By Theorem 5.1, it follows that $\Pi^n_{\mathrm{HR}} \sqcup \Pi \equiv_{\mathrm{m}} \Pi^n_{\mathrm{H}} \sqcup \Pi^n_{\mathrm{R}} \sqcup \Pi$ for any Π such that the joins are defined. For example, such a context is a module generating all symmetric directed graphs of n nodes defined as $\Pi^n_{\mathrm{G}} = \langle G^n, \emptyset, I, \emptyset \rangle$, where

$$G^n = \{\{\mathsf{arc}(x,y)\}. \mid 1 \leq x, y \leq n\} \cup \{\mathsf{arc}(y,x) \leftarrow \mathsf{arc}(x,y). \mid 1 \leq x, y \leq n\}. \quad \blacksquare$$

There is an alternative formulation for modular equivalence taking features from strong equivalence [78]. Modules $\Pi_1 = \langle R_1, I_1, O_1, H_1 \rangle, \Pi_2 = \langle R_2, I_2, O_2, H_2 \rangle \in \mathcal{M}$ are *semantically* or *strongly modularly equivalent*, denoted by $\Pi_1 \equiv_{\mathrm{sm}} \Pi_2$, if and only if $I_1 = I_2$ and $\Pi_1 \sqcup \Pi \equiv_{\mathrm{v}} \Pi_2 \sqcup \Pi$ for all $\Pi \in \mathcal{M}$ such that $\Pi_1 \sqcup \Pi$ and $\Pi_2 \sqcup \Pi$ are defined [**P1**]. It is straightforward to see that $\equiv_{\mathrm{sm}}$ is a congruence for $\sqcup$. Interestingly, $\equiv_{\mathrm{sm}}$ defines exactly the same equivalence classes as $\equiv_{\mathrm{m}}$. In [**P1**, Theorem 5.7] this is shown for $\mathcal{M}_{\mathrm{s}}$, and the proof straightforwardly generalizes for $\mathcal{M}_{\mathrm{d}}$ using the module theorem in the disjunctive case.

Theorem 5.3. *$\Pi_1 \equiv_{\mathrm{m}} \Pi_2$ if and only if $\Pi_1 \equiv_{\mathrm{sm}} \Pi_2$ for any $\Pi_1, \Pi_2 \in \mathcal{M}$.*

5.2 COMPUTATIONAL COMPLEXITY AND EQUIVALENCE VERIFICATION

The task of verifying weak, strong, and uniform equivalence is a **coNP**-complete decision problem for programs in $\mathcal{C}_{\mathrm{n}}$ [105, 33], and the complexity results can be generalized for programs in $\mathcal{C}_{\mathrm{s}}$ in a similar way (for weak equivalence, see [**P3**], for instance). For programs in $\mathcal{C}_{\mathrm{d}}$, verification of strong equivalence is a **coNP**-complete decision problem [105, 81, 126] and the verification of both weak and uniform equivalence is a $\Pi^{\mathbf{P}}_2$-complete decision problem [33, 126]. Thus, with the exception of strong equivalence for $\mathcal{C}_{\mathrm{d}}$, the computational complexity of deciding $\equiv$, $\equiv_{\mathrm{s}}$, and $\equiv_{\mathrm{u}}$ is the same as that of checking the existence of stable models for the respective class of programs $\mathcal{C}$. For the case of the more refined equivalence relations and further analysis, see [40, 35, 134]. The complexity of verifying visible/modular equivalence for $\mathcal{C}_{\mathrm{s}}$ and $\mathcal{M}_{\mathrm{s}}$, respectively, is analyzed in [**P1**, **P3**]. If the use of hidden atoms is not limited in any way, the problems of verifying visible and modular equivalence become at least as hard as the *counting problem* which is #**P**-complete [127]. It is possible, however, to govern the computational complexity by limiting the use of hidden atoms by the property of having *enough visible atoms* [**P3**], which will be reviewed next.

Intuitively, if a program $R \in \mathcal{C}$ (respectively a module $\Pi \in \mathcal{M}$) has enough visible atoms, the *EVA property* for short, then each interpretation of $\mathrm{At}_{\mathrm{v}}(R)$ (respectively $\mathrm{At}_{\mathrm{v}}(\Pi)$) *uniquely* determines an interpretation of $\mathrm{At}_{\mathrm{h}}(R)$ (respectively $\mathrm{At}_{\mathrm{h}}(\Pi)$). Consequently, the stable models can be distinguished on the basis of their visible parts and the projective equivalence defined in (5.1) coincides with visible equivalence under the EVA assumption. For the purpose of a formal definition for the EVA property, the *hidden part* of a module $\Pi = \langle R, I, O, H \rangle \in \mathcal{M}$ is defined as a module $\Pi_{\mathrm{h}} = \langle R_{\mathrm{h}}, I \cup O, H, \emptyset \rangle \in \mathcal{M}$ where R_{h} contains the rules defining the hidden atoms, that is,

- a *projected* choice rule

$$\{a'_1, \ldots, a'_k\} \leftarrow b_1, \ldots, b_n, {\sim}c_1, \ldots, {\sim}c_m$$

for each choice rule (3.4) in R such that

$$\{a'_1, \ldots, a'_k\} = \{a_1, \ldots, a_h\} \cap H \neq \emptyset;$$

- a weight rule

$$a \leftarrow w \leq \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}, {\sim}c_1 = w_{c_1}, \ldots, {\sim}c_m = w_{c_m}\}$$

for each weight rule (3.3) in R such that $a \in H$; and

- a *shifted* disjunctive rule

$$a'_1 \vee \cdots \vee a'_k \leftarrow b_1, \ldots, b_n, {\sim}c_1, \ldots, {\sim}c_m, {\sim}a''_1, \ldots, {\sim}a''_l$$

for each disjunctive rule (3.9) in R such that

$$\begin{array}{rcl} \{a'_1, \ldots, a'_k\} & = & \{a_1, \ldots, a_h\} \cap H \neq \emptyset, \text{ and} \\ \{a''_1, \ldots, a''_l\} & = & \{a_1, \ldots, a_h\} \setminus H. \end{array}$$

It is worth noticing that choice rules and disjunctive rules require a different treatment. The semantics of choice rules is such that there is no minimality assumption for the head atoms. Thus, it suffices simply to project the head with respect to the hidden atoms. The disjunctive case is more involved. The semantics of a disjunction in the head is based on minimality, and it is necessary to *shift* the visible atoms from the head [**P2**].

Definition 5.4. *A module $\Pi = \langle R, I, O, H\rangle \in \mathcal{M}$ has the EVA property if and only if the hidden part $\Pi_{\mathrm{h}} = \langle R_{\mathrm{h}}, I \cup O, H, \emptyset\rangle$ has a unique stable model M for each $N \subseteq \mathrm{At_v}(\Pi) = I \cup O$ such that $M \cap (I \cup O) = N$.*

The definition above also covers the case of ordinary, input-free logic programs, since any $R \in \mathcal{C}$ can be viewed as a module

$$\Pi = \langle R, \emptyset, \mathrm{At_v}(R), \mathrm{At_h}(R)\rangle \in \mathcal{M}.$$

It is always possible to enforce the EVA property by uncovering sufficiently many hidden atoms. A module Π for which $\mathrm{At_h}(\Pi) = \emptyset$ has clearly enough visible atoms because Π_{h} has no rules. It is also important to realize that choice rules and disjunctive rules involving hidden atoms in their heads are likely to break up the EVA property unless additional constraints are introduced to exclude multiple models created by choices and disjunctions. Although verifying the EVA property can be hard in general, see [**P3**, Proposition 4.14], there are syntactic classes of logic programs which are guaranteed to have enough visible atoms and no computational efforts are needed to verify this, for instance, modules $\Pi = \langle R, I, O, H\rangle$ for which R_{h} is positive or *stratified* [4] in some sense.

Assuming the EVA property, the verification of visible equivalence is a **coNP**-complete decision problem for $\mathcal{C}_{\mathrm{s}}$ [**P3**], and the verification of modular equivalence is a **coNP**-complete decision problem for $\mathcal{M}_{\mathrm{s}}$ [**P1**]. For

modules in $\mathcal{M}_{\mathrm{d}}$ with the EVA property, the verification of modular equivalence can be reduced to the problem of finding stable models for $\mathcal{M}_{\mathrm{d}}$ by combining the ideas in the translations in [101] (for capturing a counter-example for the equivalence) and [**P4**] (for computing the unique stable model for the hidden part guaranteed by the EVA property). $\Pi_2^{\mathbf{P}}$-hardness follows from the observation that the $\Pi_2^{\mathbf{P}}$-complete problem of deciding $R_1 \equiv R_2$ for $R_1, R_2 \in \mathcal{C}_{\mathrm{d}}$ [126, 33] reduces to deciding

$$\langle R_1, \emptyset, \mathrm{At}(R_1) \cup \mathrm{At}(R_2), \emptyset \rangle \equiv_{\mathrm{m}} \langle R_2, \emptyset, \mathrm{At}(R_1) \cup \mathrm{At}(R_2), \emptyset \rangle.$$

Thus, assuming the EVA property, the verification of modular equivalence is a $\Pi_2^{\mathbf{P}}$-complete decision problem for $\mathcal{M}_{\mathrm{d}}$. Due to the close connection of modular and visible equivalence, also deciding visible equivalence of programs in $\mathcal{C}_{\mathrm{d}}$ with the EVA property is a $\Pi_2^{\mathbf{P}}$-complete problem.

The computational complexity of deciding equivalence has motivated several *translation-based* approaches for the equivalence verification task. Overall the idea is to translate the problem of verifying equivalence of two programs into another problem for which there already exist efficient solvers. In [66] verification of weak equivalence of two SMODELS programs is reduced to the existence of stable models by introducing a translation $\mathrm{EQT} : \mathcal{C}_{\mathrm{s}} \times \mathcal{C}_{\mathrm{s}} \to \mathcal{C}_{\mathrm{s}}$ such that given programs $R_1, R_2 \in \mathcal{C}_{\mathrm{s}}$, $R_1 \equiv R_2$ if and only if $\mathrm{SM}(\mathrm{EQT}(R_1, R_2)) = \emptyset$ and $\mathrm{SM}(\mathrm{EQT}(R_2, R_1)) = \emptyset$. The tool LPEQ [67] implements the translation, and also covers verification of strong and classical equivalence for programs in $\mathcal{C}_{\mathrm{n}}$. Similar approach is followed in [101], and the tool DLPEQ implements the translation for verifying weak equivalence of programs in $\mathcal{C}_{\mathrm{d}}$.

The tool SELP [23] reduces the problem of deciding strong equivalence of disjunctive logic programs into propositional satisfiability. The tool SE-TEST [31] reduces the problem of verifying strong equivalence of disjunctive *first-order Datalog programs* under the stable model semantics to the unsatisfiability of *Bernays-Schönfinkel formulas*. The tool CCT implements the general framework for specifying program correspondences introduced in [40] by reducing the correspondence problem to a *quantified Boolean formula* (QBF) such that the resulting QBF evaluates to true if and only if the correspondence problem holds [125]. In the special case of uniform equivalence relative to a context set and a projection set, CCT uses an encoding proposed in [99] and it can generate QBFs for computing *counter-examples* in addition to solving the decision problem whether correspondence holds.

In [**P3**] the translation-based approach from [66] is extended to cover the verification of visible equivalence of SMODELS programs with the EVA property. In [**P4**] the translation-based approach is adjusted to the verification of modular equivalence of modules in $\mathcal{M}_{\mathrm{s}}$ with the EVA property. Interestingly, the congruence property of modular equivalence and the compositionality of the stable model semantics allow further refinements in terms of modularization. In the next theorem $\mathrm{EQT} : \mathcal{M}_{\mathrm{s}} \times \mathcal{M}_{\mathrm{s}} \to \mathcal{M}_{\mathrm{s}}$ refers to the translation function presented in [**P4,** Definition 10].

Theorem 5.5. *Let* $\Pi_1 = \langle R_1, I, O, H_1 \rangle, \Pi_2 = \langle R_2, I, O, H_2 \rangle \in \mathcal{M}_{\mathrm{s}}$ *be modules with the EVA property, and* Π *any module in* $\mathcal{M}_{\mathrm{s}}$ *such that* $\Pi_1 \sqcup \Pi$ *and* $\Pi_2 \sqcup \Pi$ *are defined. Then* $\Pi_1 \sqcup \Pi \equiv_{\mathrm{m}} \Pi_2 \sqcup \Pi$ *if and only if both* $\mathrm{SM}(\mathrm{EQT}(\Pi_1, \Pi_2) \sqcup \Pi) = \emptyset$ *and* $\mathrm{SM}(\mathrm{EQT}(\Pi_2, \Pi_1) \sqcup \Pi) = \emptyset$.

Theorem 5.5 shows that there is no need to translate the common context Π. This is of significance, as the length of the translation $\mathrm{EQT}(\Pi_1, \Pi_2) \sqcup \Pi$ involved in the equivalence verification task might be close to half of that of $\mathrm{EQT}(\Pi_1 \sqcup \Pi, \Pi_2 \sqcup \Pi)$, if Π is large. By combining the ideas from [101] and [**P4**], Theorem 5.5 can be generalized in a rather straight-forward manner for $\mathcal{M}_{\mathrm{d}}$. Moreover, a stable model for the translation $\mathrm{EQT}(\Pi_1, \Pi_2)$ captures a counter-example for the equivalence $\Pi_1 \equiv_{\mathrm{m}} \Pi_2$. In [**P4,** Example 6] it is illustrated how a counter-example can be extracted from a stable model of the translation in practice.

Furthermore, Theorem 5.5 enables modularization of equivalence verification [**P4**]. If $\Pi_2 \in \mathcal{M}$ is obtained from $\Pi_1 \in \mathcal{M}$ through local modifications, it is likely that there is a partitioning for Π_1 and Π_2 such that $\Pi_1 = \Pi_1^1 \sqcup \cdots \sqcup \Pi_1^n$ and $\Pi_2 = \Pi_2^1 \sqcup \cdots \sqcup \Pi_2^n$ where Π_1^i is compatible with Π_2^i for all i in the sense that they have the same input/output interface. If $\Pi_1^i \equiv_{\mathrm{m}} \Pi_2^i$ holds and their equivalence can be verified efficiently, then Theorem 5.1 implies that Π_1^i and Π_2^i are modularly equivalent in every possible context. If this is not the case, it is still possible to organize the verification of $\Pi_1 \equiv_{\mathrm{m}} \Pi_2$ as a sequence of n module-level tests as follows:

$$\Pi_1^i \sqcup \Pi_i \equiv_{\mathrm{m}} \Pi_2^i \sqcup \Pi_i \tag{5.4}$$

where $1 \leq i \leq n$, and in each test (5.4) modules differ in Π_1^i and Π_2^i for which the other modules form a common context

$$\Pi_i = (\bigsqcup_{j=1}^{i-1} \Pi_2^j) \sqcup (\bigsqcup_{j=i+i}^{n} \Pi_1^j).$$

6 PROPERTIES OF TRANSLATION FUNCTIONS

Transformations between program classes provide means for an expressiveness analysis, see [98, 47, 39, 61, 63, 64, 79, 104], for instance. The properties of transformations are often presented in terms of translation time, model correspondence with respect to an equivalence relation, and modularity of the translation in some sense.

Niemelä [98] defines that a translation is *modular*, if adding a set of facts to a program leads to a local change not involving the rest of the translation, and shows that there is no modular translation from logic programs under the stable model semantics to propositional satisfiability of a set of clauses.

Ferraris [47] defines that a translation $\mathrm{Tr} : \mathcal{C}_1 \rightarrow \mathcal{C}_2$ between program classes $\mathcal{C}_1$ and $\mathcal{C}_2$ is modular if for every rule $r \in \mathcal{C}_1$ it holds $\mathrm{At}(\mathrm{Tr}(r)) \subseteq \mathrm{At}(r)$, and *sound* if for every $R \in \mathcal{C}_1$, it holds $R \equiv \cup_{r \in R} \mathrm{Tr}(r)$. Thus a modular transformation does not introduce any auxiliary atoms, and a sound transformation preserves the stable models. Ferraris shows that every transformation $\mathrm{Tr} : \mathcal{C}_1 \rightarrow \mathcal{C}_2$ such that $\mathcal{C}_1$ contains all *unary rules* is sound if and only if for each rule $r \in \mathcal{C}_1$, $r \equiv_{\mathrm{s}} \mathrm{Tr}(r)$ and analyzes expressiveness of different program classes in terms of existence of a sound transformation.

Janhunen [64] proposes an analysis method which is based on the existence of *polynomial*, *faithful* and *modular* (PFM) translation functions between classes of logic programs, and shows that these three properties are preserved under compositions of programs in terms of *disjoint unions* [64]. Intuitively, a faithful translation preserves the roles of visible atoms in the transformation, and enforces bijective correspondence between the models of the original program and its translation based on visible equivalence. A translation is polynomial if it can be computed in polynomial time with respect to the length of the original program in symbols. Programs $R, R' \in \mathcal{C}$ satisfy *module conditions* if and only if (i) $R \cap R' = \emptyset$, (ii) $\mathrm{At}_{\mathrm{v}}(R) = \mathrm{At}_{\mathrm{v}}(R')$, and (iii) $\mathrm{At}_{\mathrm{h}}(R) \cap \mathrm{At}(R') = \emptyset$ and $\mathrm{At}(R) \cap \mathrm{At}_{\mathrm{h}}(R') = \emptyset$. A translation $\mathrm{Tr} : \mathcal{C}_1 \rightarrow \mathcal{C}_2$ is modular if and only if for all $R, R' \in \mathcal{C}$ satisfying the module conditions, $\mathrm{Tr}(R)$ and $\mathrm{Tr}(R')$ satisfy the module conditions and $\mathrm{Tr}(R \cup R') = \mathrm{Tr}(R) \cup \mathrm{Tr}(R')$.

In [**P5**] the expressive power of *parallel circumscription* and positive disjunctive logic programs under the stable models semantics is compared using the existence of a polynomial, faithful and modular translation function in terms of [64] as a criterion, and in particular, it is shown that there is no PFM-translation from parallel circumscription to positive disjunctive logic programs [**P5**, Theorem 2].

Eiter and Polleres [39] provide a translation from propositional *head-cycle-free* [13] extended disjunctive logic programs to disjunctive logic programs which integrates the guess and check programs into a single program solving the problem. The properties of the translation studied in [39] differ from those in [64] at first glance, but the translation in [39] is polynomial and modular in the sense of [64]. However, as a result of different design criteria, the translation in [39] is not faithful in the sense of [64].

Pearce et al. [104] also consider PFM-translation functions, and introduce a polynomial, strongly faithful and modular translation from nested logic pro-

grams to disjunctive logic programs. However, the conditions of faithfulness and modularity differ slightly from those in [64]. Pearce et al. [104] define that a translation $\mathrm{Tr} : \mathcal{C}_1 \to \mathcal{C}_2$ is faithful if and only if $\mathrm{SM}(R) = \{M \cap \mathrm{At}(R) \mid M \in \mathrm{SM}(\mathrm{Tr}(R))\}$. Thus instead of visible equivalence, a projective variant is considered and the exact number of stable models needs not to be preserved. Moreover, they define that a translation is *strongly faithful* if and only if for all programs $R, R' \in \mathcal{C}_1$,

$$\mathrm{SM}(R \cup R') = \{M \cap \mathrm{At}(R \cup R') \mid M \in \mathrm{SM}(\mathrm{Tr}(R) \cup R')\}.$$

Finally, a translation is modular if and only if $\mathrm{Tr}(R \cup R') = \mathrm{Tr}(R) \cup \mathrm{Tr}(R')$ for all $R, R' \in \mathcal{C}_1$. Thus the condition is a stronger variant of that by Janhunen [64].

Lifschitz et al. [79] present a translation from nested logic programs to extended disjunctive logic programs. Their translation does not introduce new atoms and preserves strong equivalence. However, the translation is exponential in the worst case. Inoue and Sakama [61] present two transformations from *general extended disjunctive program* to extended disjunctive logic programs, that is, translations to remove default negation from the heads of rules. Both translations are polynomial. The one introducing new atoms is modular and faithful. The other is based on shifting, and an additional stability condition on minimal models is needed to achieve faithfulness. Janhunen [63] considers disjunctive programs without classical negation and presents another PFM-translation to remove default negation from the heads of rules. In contrast to the translation by Inoue and Sakama [61], his translation is linear. It is worth noticing that the condition for modularity proposed in [63] slightly differs from the one considered in [64].

In [**P1**], the conditions by Janhunen [64] are generalized and slightly modified to fit the module framework [**P1, P2**]. These conditions are presented in detail in Definition 6.1. Modular equivalence is used instead of visible equivalence in the faithfulness property, and a *strongly faithful* translation preserves the roles of *all*, not just visible atoms in the original encoding. A translation is *modular*, if each module can be translated separately without affecting the outcome. A $\sqcup$*-preserving* translation is such that possible compositions of modules are not limited by the translation. For convenience, an operator $\mathsf{reveal}(\Pi, A) = \langle R, I, O \cup A, H \setminus A \rangle$ is defined to make a set of hidden atoms $A \subseteq H$ of a module $\Pi = \langle R, I, O, H \rangle \in \mathcal{M}$ visible to other modules.

Definition 6.1. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two classes of logic program modules such that $\mathcal{M}_2 \subseteq \mathcal{M}_1$. A translation function $\mathrm{Tr} : \mathcal{M}_1 \to \mathcal{M}_2$ is*

1. ***polynomial*** *if and only if for all modules $\Pi \in \mathcal{M}_1$, the translation $\mathrm{Tr}(\Pi) \in \mathcal{M}_2$ can be computed in time (and hence also space) polynomial to $\|\Pi\|$, the length of Π in symbols;*

2. ***faithful*** *if and only if for all modules $\Pi \in \mathcal{M}_1$, $\Pi \equiv_\mathrm{m} \mathrm{Tr}(\Pi)$;*

3. ***strongly faithful*** *if and only if for all modules $\Pi \in \mathcal{M}_1$,*

$$\mathsf{reveal}(\Pi, \mathrm{At_h}(\Pi)) \equiv_\mathrm{m} \mathsf{reveal}(\mathrm{Tr}(\Pi), \mathrm{At_h}(\Pi));$$

4. $\sqcup$-***preserving*** *if and only if for all modules* $\Pi, \Pi' \in \mathcal{M}_1$*, if* $\Pi \sqcup \Pi'$ *is defined, then* $\mathrm{Tr}(\Pi) \sqcup \mathrm{Tr}(\Pi')$ *is defined;*

5. ***modular*** *if and only if for all modules* $\Pi, \Pi' \in \mathcal{M}_1$ *such that* $\Pi \sqcup \Pi'$ *is defined,* $\mathrm{Tr}(\Pi) \sqcup \mathrm{Tr}(\Pi') = \mathrm{Tr}(\Pi \sqcup \Pi')$.

It is worth noticing that the condition of modularity in Definition 6.1 is different from the one used in [64]. While the module conditions (i) and (iii) make sense with respect to $\sqcup$, condition (ii) is problematic. If (ii) is brought to a modular setting in the sense of [**P1, P2**], then $\mathrm{At_v}(R) = \mathrm{At_v}(R')$ implies that the respective join of modules $\langle R, \emptyset, \mathrm{At_v}(R), \mathrm{At_h}(R)\rangle$ and $\langle R', \emptyset, \mathrm{At_v}(R'), \mathrm{At_h}(R')\rangle$ is not defined. Thus, the counter-example presented in the proof of [**P5**, Theorem 2] does not work when modular translation is defined as in Definition 6.1. Also, the condition for strong faithfulness differs from the one by Pearce et al. [104]. The condition used in Definition 6.1 resembles the faithfulness criterion in [104] but bijective model correspondence is insisted by modular equivalence.

Furthermore, in some cases a weaker variant of the condition for modularity might be of interest, that is, one may consider modular equivalence

$$\mathrm{Tr}(\Pi) \sqcup \mathrm{Tr}(\Pi') \equiv_{\mathrm{m}} \mathrm{Tr}(\Pi \sqcup \Pi')$$

instead of the syntactical equivalence used in Definition 6.1.

The module theorem (Theorem 4.7) lends itself to extensions for further classes of logic program modules which can be brought into effect in terms of strongly faithful, $\sqcup$-preserving, and modular translations for the removal of new syntax [**P1**]. More formally, given two classes of logic program modules $\mathcal{M}_1, \mathcal{M}_2$ such that $\mathcal{M}_2 \subseteq \mathcal{M}_1$, and the module theorem holds for $\mathcal{M}_2$, if there is a strongly faithful, $\sqcup$-preserving, and modular translation from $\mathcal{M}_1$ to $\mathcal{M}_2$, then the module theorem holds for $\mathcal{M}_1$, too. In [**P1**] this is demonstrated by introducing such a translation from $\mathcal{M}_{\mathrm{s}}$ to $\mathcal{M}_{\mathrm{n}}$. It is worth noticing that the translation time is of no interest in this case.

Niemelä et al. [118] propose a linear translation function which embeds the class of *weight constraint programs* into $\mathcal{C}_{\mathrm{s}}$. Since each weight constraint and each rule is translated independently, the translation is modular. In [**P3**] the translation from [118] is shown to be faithful. From faithfulness of the transformation it directly follows that the translation-based method for verification of visible equivalence of SMODELS programs proposed in [**P3**] covers also the case of weight constraint programs through the translation.

As another example of transformations between programs classes, translations for embedding parallel/prioritized circumscription into answer set programming are discussed in the next section.

6.1 EMBEDDING CIRCUMSCRIPTION INTO ASP

In this section, *positive* programs in $\mathcal{C}_{\mathrm{d}}$ are considered. The stable model semantics of disjunctive logic programs is based on minimal models which makes every atom appearing in a logic program false by default. While this feature is highly useful and leads to concise encodings of problems as programs, it occasionally makes knowledge representation with rules difficult.

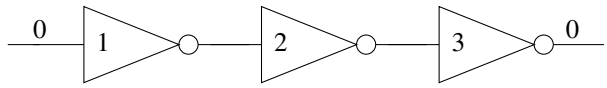

Figure 6.1: The circuit from Example 6.2 consisting of three inverters connected in series together with observations.

Example 6.2. A digital circuit can be modelled using propositional theories following the ideas from [12]. For instance, an inverter x is described by a propositional theory $(\mathsf{out}(x) \leftrightarrow \neg\mathsf{in}(x)) \vee \mathsf{ab}(x)$, where the atoms $\mathsf{in}(x)$ and $\mathsf{out}(x)$ model the interconnection of the input and the output of x, respectively, and $\mathsf{ab}(x)$ expresses the fact that x is operating against its specification. This theory can be equivalently formulated as a positive disjunctive logic program,

$$I(x) = \{\mathsf{ab}(x) \leftarrow \mathsf{in}(x), \mathsf{out}(x).\ \mathsf{in}(x) \vee \mathsf{out}(x) \vee \mathsf{ab}(x).\}.$$

This way of thinking carries over to larger circuits which have also other gates than inverters as their components, and the description of the circuit can be formed component-by-component.

Consider for instance the circuit C presented in Figure 6.1. The circuit C consists of three inverters $\{1, 2, 3\}$ connected in series together with observations $\neg\mathsf{in}(1)$ and $\neg\mathsf{out}(3)$ indicating a faulty behavior of the circuit.

The inverters are modelled using $I(1) \cup I(2) \cup I(3)$, and the wiring of the circuit C can be specified as

$$CW(C) = \{\mathsf{in}(2) \leftarrow \mathsf{out}(1).\ \mathsf{out}(1) \leftarrow \mathsf{in}(2).\ \mathsf{in}(3) \leftarrow \mathsf{out}(2).\ \mathsf{out}(2) \leftarrow \mathsf{in}(3).\}.$$

Finally, in order to obtain diagnoses for circuit C, the specification of the circuit is combined with the observations,

$$R(C) = I(1) \cup I(2) \cup I(3) \cup CW(C) \cup \{\bot \leftarrow \mathsf{in}(1).\ \bot \leftarrow \mathsf{out}(3).\}.$$

However, the stable models of the specification do not correspond to Reiter-style minimal diagnoses [109]. For instance, $R(C)$ has four stable models,

$$\begin{aligned} M_1 &= \{\mathsf{in}(2), \mathsf{out}(1), \mathsf{ab}(3)\}, \\ M_2 &= \{\mathsf{in}(3), \mathsf{out}(2), \mathsf{ab}(1)\}, \\ M_3 &= \{\mathsf{in}(2), \mathsf{out}(1), \mathsf{in}(3), \mathsf{out}(2), \mathsf{ab}(2)\}, \text{ and} \\ M_4 &= \{\mathsf{ab}(1), \mathsf{ab}(2), \mathsf{ab}(3)\}, \end{aligned}$$

but M_4 does not correspond to a minimal diagnosis, as all three inverters are faulty according to it. Similar spurious minimal models are also obtained for more complex circuits encoded in this way. ■

Lifschitz' *parallel circumscription* [74] allows a more refined control for minimality, in which certain atoms are allowed to *vary* or to have *fixed* values while others are falsified as far as possible. In particular, varying atoms are interesting as they enhance the knowledge representation capability over ordinary circumscription by McCarthy [94]. *Prioritized circumscription* [74] generalizes the setting of parallel circumscription in terms of priority classes for atoms being minimized.

Definition 6.3. *Let R be a positive program in $\mathcal{C}_{\mathrm{d}}$ and let $P, V, F \subseteq \mathrm{At}(R)$ be* disjoint *sets of atoms such that $\mathrm{At}(R) = P \cup V \cup F$. A model $M \models R$ is $\langle P, V, F\rangle$-minimal if and only if there is no $N \models R$ such that*

(i) $N \cap P \subset M \cap P$ *and*

(ii) $N \cap F = M \cap F$.

By this definition, atoms in P are subject to minimization, that is, falsified as far as possible, while the truth values of atoms in V may vary freely and the truth values of atoms in F are kept fixed. Note that in the notation for $\langle P, V, F\rangle$-minimality one of the sets P, V, and F is redundant, as given any two, the third one is implicitly clear from the context. The conventional case that all atoms are subject to minimization is covered by considering the $\langle \mathrm{At}(R), \emptyset, \emptyset\rangle$-minimal models of a positive disjunctive logic program R.

Example 6.4. Recall program $R(C)$ representing the behavior of circuit C from Example 6.2. By defining

$$\begin{aligned} P &= \{\mathsf{ab}(1), \mathsf{ab}(2), \mathsf{ab}(3)\} \text{ and} \\ V &= \{\mathsf{in}(1), \mathsf{in}(2), \mathsf{in}(3), \mathsf{out}(1), \mathsf{out}(2), \mathsf{out}(3)\}, \end{aligned}$$

the minimal diagnoses for circuit C correspond exactly to $\langle P, V, \emptyset\rangle$-minimal models of $R(C)$, namely M_1, M_2, and M_3. ■

Let $\mathsf{circ}(R, P, V, F)$ denote the parallel circumscription of a positive disjunctive programs R. An extended notation $\mathsf{circ}(R, P_1 > \ldots > P_k, V, F)$ is introduced to represent the prioritized circumscription of R which includes the parallel circumscription of R as its special case, that is, when $k = 1$. The idea is that atoms in P_1 are falsified with the highest priority, those in P_2 with the next highest priority, and so on. Lifschitz [74] shows that a prioritized circumscription $\mathsf{circ}(R, P_1 > \ldots > P_k, V, F)$ corresponds to a conjunction

$$\bigwedge_{i=1}^{k} \mathsf{circ}(R, P_i, P_{i+1} \cup \ldots \cup P_k \cup V, P_1 \cup \ldots \cup P_{i-1} \cup F). \tag{6.1}$$

The conjunction (6.1) does not have a direct interpretation as a disjunctive logic program but such a representation can be obtained using a translation from [**P6**]. A drawback is that, roughly speaking, $2k$ copies of R must be created which gives a quadratic nature for the overall transformation proposed in [**P6**] because $k \leq |\mathrm{At}(R)| \leq \|R\|$, that is, the length of R in symbols.

Wakaki and Inoue [130] generalize Definition 6.3 to the case of prioritized circumscription.

Definition 6.5. *A model $M \models R$ for a positive disjunctive logic program $R \in \mathcal{C}_{\mathrm{d}}$ is $\langle P_1 > \ldots > P_k, V, F\rangle$-minimal if and only if there is no $N \models R$ such that*

(i) $N \cap (P_1 \cup \ldots \cup P_{i-1}) = M \cap (P_1 \cup \ldots \cup P_{i-1})$ *and* $N \cap P_i \subset M \cap P_i$ *for some* $1 \leq i \leq k$*; and*

(ii) $N \cap F = M \cap F$.

Example 6.6. Prioritized circumscription allows for further refinements of the diagnoses obtained for circuit C in Example 6.4. For instance, consider priorities $\{\mathsf{ab}(1)\} > \{\mathsf{ab}(2)\} > \{\mathsf{ab}(3)\}$ for P, that is, first the atom $\mathsf{ab}(1)$ is falsified, then $\mathsf{ab}(2)$, and finally $\mathsf{ab}(3)$. The last minimization fails, however, as $R(C) \cup \{\neg\mathsf{ab}(1), \neg\mathsf{ab}(2)\} \models \mathsf{ab}(3)$. Thus, M_1 is the unique $\langle\{\mathsf{ab}(1)\} > \{\mathsf{ab}(2)\} > \{\mathsf{ab}(3)\}, V, \emptyset\rangle$-minimal model of $R(C)$. ■

Since the stable models semantics does not incorporate support to varying atoms or priorities for minimization, the current (disjunctive) ASP solvers have no support for them either. An efficient embedding from parallel and prioritized circumscription into disjunctive logic programs could enable using the current ASP tools for computing circumscription.

De Kleer and Konolige [26] consider parallel circumscription in the first order case, and present a basic technique for eliminating fixed predicates based on a linear, faithful, and modular transformation. The case of varying predicates is addressed by Cadoli et al. [22] but since they consider circumscription in the first order logic, a query-based equivalence rather than an exact model correspondence is of their interest.

In addition to these general results, there is a number of attempts to reduce parallel/prioritized circumscription into disjunctive logic programming under the stable model semantics. Gelfond and Lifschitz [54] address prioritized circumscription but their translation scheme is applicable to *stratified* circumscriptive theories only. The translation of parallel circumscription presented by Sakama and Inoue [112] is based on *characteristic clauses* which implies an exponential space and time complexity in the worst case. In [113], the same authors embed prioritized circumscription into *prioritized logic programs* based on a different semantics. Lee and Lin [71] characterize parallel circumscription in terms of *loop formulas* and exploit them to obtain an embedding to disjunctive logic programming. However, the number of loops can be exponential in the worst case. Thus, it remains open whether an efficient translation is feasible in general using their approach.

Wakaki and Inoue [130] concentrate on prioritized circumscription and design a two-phase procedure for the computation of minimal models. The first phase generates model candidates which are then tested for minimality in the sense of prioritized circumscription. Both the model generator and the tester are represented as *separate* disjunctive logic programs. There is an implementation of the procedure, named CIRCUM1, but it is rather inefficient since all model candidates are computed first. Wakaki and Tomita [131] improve the procedure by Wakaki and Inoue [130] and integrate the generating and testing programs into one. However, this is not a one-shot transformation because the answer sets of the generating program have to be computed and counted before the testing part can be really created. The resulting implementation, CIRCUM2, appears to be faster than CIRCUM1, but it consumes an exponential space in the worst case.

Despite that the analysis in [**P5**] reveals that varying atoms are of global nature at least to some extent, it is possible to embed circumscription into disjunctive logic programming in a faithful way. In [**P5**] the first *linear* and *faithful* translation from parallel circumscription into disjunctive logic programs with negation is proposed. The approach is influenced by the generate-and-test architecture of GNT [65] and it combines the generating and testing pro-

grams in one disjunctive logic program. The fixed atoms are assumed to have been removed using the translation by De Kleer and Konolige [26]. Then the $\langle P, V, \emptyset \rangle$-minimality for a model M of a positive logic program $R \in \mathcal{C}_{\mathrm{d}}$ is characterized in terms of propositional satisfiability. The idea is to check whether a set of disjunctive rules $\mathrm{Tr}_{\mathrm{UNSAT}}(R, P, M)$ is unsatisfiable in the classical sense [**P5**]. The propositional unsatisfiability check is then encoded with the primitives of ASP using the approach by Eiter and Gottlob [36]. The faithfulness of the translation guarantees that the $\langle P, V, \emptyset \rangle$-minimal models M of positive $R \in \mathcal{C}_{\mathrm{d}}$ and the stable models N of its translation are in a bijective relationship such that $M = N \cap \mathrm{At}(R)$ holds for each pair of models.

In [**P7**], the approach proposed in [**P5**] is generalized to the case of prioritized circumscription. The module architecture proposed in [**P2**] allows representing the translation $\mathrm{Tr}_{\mathrm{circ2dlp}}(R, P_1 > \cdots > P_k, V, F)$ as a join of two modules, one generating model candidates while the other is performing a minimality check. The compositionality of the stable model semantics formalized by the module theorem, simplifies argumentation for the correctness of the translation compared to that in [**P5**].

The implementation of the transformation called CIRC2DLP [**P6, P7**] shows a promising performance compared to that of CIRCUM2 [131], and enables the systematic use of parallel and prioritized circumscription as a primitive in disjunctive logic programming for developing more compact formulations of problems as logic programs.

7 CONCLUSIONS

A simple and intuitive notion of a logic program module that interacts with other modules through a well-defined input/output interface is introduced for SMODELS programs and disjunctive logic programs [**P1, P2**], respectively. The design has its roots in a module architecture proposed for conventional logic programs [48], but the architecture is tailored to better meet the needs of answer set programming. Perhaps the most important objective in this respect is to achieve the compositionality of the stable model semantics, that is, the semantics of an entire program should depend directly on the semantics assigned to its modules. This main result is formalized as the *module theorem* [**P1, P2**] which links program-level stability with module-level stability. The theorem holds under the assumption that positively interdependent atoms are always placed in the same module. The join operation $\sqcup$ defined for program modules effectively formalizes this constraint, and the conditions under which the join of two modules is defined can be viewed as a reflection of good programming style in answer set programming.

The module theorem is also a proper generalization of the splitting set theorem [80]. The main difference is that splitting sets do not enable any kind of recursion between modules. Even though the module theorem is proved to demonstrate the feasibility of the respective module architecture, it is also applied as a tool to simplify mathematical proofs [**P1, P2, P4, P7**]. Moreover, in [**P1**] it is shown that the module theorem can be extended for further classes of logic programs provided that there is a strongly faithful, $\sqcup$-preserving, and modular translation to one of classes for which the module theorem holds. This strategy is used to generalize the module theorem originally presented for normal logic program modules to cover the class of SMODELS program modules [**P1**].

The second main theme is the notion of modular equivalence [**P1, P2, P4**] which is proved to be a proper congruence relation for program composition using $\sqcup$. Thus, modular equivalence is preserved under substitutions of modularly equivalent program modules. Since weak equivalence is not a congruence for union of programs but strong equivalence is by definition, modular equivalence can be viewed as a reasonable compromise between these two extremes. For modules having enough visible atoms so that their stable models can be distinguished from each other on the basis of visible atoms only, that is, for modules with the EVA property, deciding modular equivalence has the same computational complexity as deciding the existence of stable models. This allows one to use an ASP solver for the actual verification task.

In order to enable the task of equivalence verification using the existing ASP solvers, a translation-based approach for verifying the visible equivalence of logic programs under the stable model semantics is proposed in [**P3**]. The translation $\mathrm{EQT}(R_1, R_2)$ and its implementation LPEQ cover the types of rules supported by the SMODELS search engine. In [**P4**] this translation-based approach is adjusted to the task of verifying modular equivalence. Moreover, in cases where two modules Π_1 and Π_2 are placed in a common context Π, the verification method can be further streamlined and it is not

necessary to translate Π. If the context Π is large, then the length of the translation $\mathrm{EQT}(\Pi_1, \Pi_2) \sqcup \Pi$ involved in the equivalence verification task is roughly half of that of $\mathrm{EQT}(\Pi_1 \sqcup \Pi, \Pi_2 \sqcup \Pi)$.

The main result in [**P5**] is a linear translation from parallel circumscription into disjunctive logic programs such that a bijective correspondence between the $\langle P, V, F\rangle$-minimal models of a positive disjunctive logic programs R and the stable models of the respective translation is obtained. In terms of the expressiveness analysis in [**P5**], the translation function is non-modular which reflects the global nature of varying atoms.

A transformation from prioritized circumscription to disjunctive logic programs is proposed in [**P7**]. The translation function generalizes and improves respective translations in [**P5, P6**], and has a distinctive combination of features. Arbitrary propositional theories R subject to prioritized circumscription are covered, and the translation can be produced in linear time and space before computing any models for it. The $\langle P_1 > \ldots > P_k, V, F\rangle$-minimal models of a positive disjunctive program R and the stable models of its translation are in a bijective relationship, while the signature $\mathrm{At}(R)$ is preserved under the translation. In contrast, all previous approaches lack some of these features. The implementation of translations in [**P5, P7**] enables the systematic use of parallel and prioritized circumscription as a primitive in disjunctive logic programming for developing more compact formulations of problems as logic programs.

7.1 TOPICS FOR FURTHER RESEARCH

The current input language of ASP is based on logic programs, and does not have support for the additional input/output interface of logic program modules. By extending the language to allow explicit module interface definitions, the module system proposed in [**P1, P2**] provides a basis for a programming-in-the-large approach to answer set programming. Moreover, the current ASP solvers have little support for modularity, and workarounds are needed to simulate modular ASP using the currents tools. Incorporating support for modularity into the solvers and grounders is crucial in order to further promote modular answer set programming.

In this thesis propositional logic program modules are considered. In practice it is, however, undesirable to always have to consider ground instances of modules. A crucial step further is to extend the concept of modularity to the non-ground case, that is, to consider program modules involving variables. Several aspects need to be considered when extending the module system to the non-ground case. For example, how to define the input/output interface and interaction of modules (using predicates, grounded atoms or constants), how to define the concept of modular equivalence for non-ground modules and still maintain the essential congruence property with respect to join of modules, and how to ground a stand-alone module.

The class of logic programs supported by the module architecture should be extended to cover even more general classes of logic programs, such as nested logic programs [79] and weight constraint programs [118], for instance. One possibility is to introduce a strongly faithful, $\sqcup$-preserving and

modular transformation [**P1**] to one of the classes of programs already covered in this work. The translation from weight constraint programs to SMODELS programs by Simons et al. [118] and the translations from nested logic programs to disjunctive logic programs by Lifschitz et al. [79] and Pearce et al. [104], are good candidates with this respect.

The global nature of varying atoms in parallel circumscription is not as definite as it first seemed based on the analysis in [**P5**]. The question how far parallel/prioritized circumscription can be modularized still remains open, however. A further goal is the generalization of stable models with prioritized minimization of models. In fact, the design of the implementation CIRC2DLP already includes a support for negative body literals in rules, which enables the computation of $\langle P_1 > \ldots > P_k, V, F\rangle$-*stable models* M of an arbitrary (not just positive) disjunctive logic program R based on the reduct R^M.

BIBLIOGRAPHY

[1] Luigia Carlucci Aiello and Fabio Massacci. Verifying security protocols as planning in logic programming. *ACM Transactions on Computational Logic*, 2(4):542–580, 2001.

[2] José Júlio Alferes and João Leite, editors. *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 2004, Proceedings*, volume 3229 of *Lecture Notes in Artificial Intelligence*. Springer, 2004.

[3] Christian Anger, Martin Gebser, Thomas Linke, André Neumann, and Torsten Schaub. The Nomore++ system. In Baral et al. [10], pages 422–426.

[4] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In Minker [96], pages 89–148.

[5] Tuomas Aura, Matt Bishop, and Dean Sniegowski. Analyzing single-server network inhibition. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW'00), 3–5 July 2000, Cambridge, England, UK*, page 108. IEEE Computer Society Press, 2000.

[6] Marcello Balduccini, Michael Gelfond, Richard Watson, and Monica Nogueira. The USA-advisor: A case study in answer set planning. In Eiter et al. [32], pages 439–442.

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

[8] Chitta Baral, Karen Chancellor, Nam Tran, Nhan Tran, Anna M. Joy, and Michael E. Berens. A knowledge based approach for representing and reasoning about signaling networks. *Bioinformatics*, 20(Suppl. 1):i15–i22, 2004.

[9] Chitta Baral, Juraj Dzifcak, and Hiro Takahashi. Macros, macro calls and use of ensembles in modular answer set programming. In Sandro Etalle and Mirosław Truszczyński, editors, *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 2006, Proceedings*, volume 4079 of *Lecture Notes in Computer Science*, pages 376–390. Springer, 2006.

[10] Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors. *Logic Programming and Nonmonotonic Reasoning, 8th International Conference, LPNMR 2005, Diamante, Italy, September 5-8, 2005, Proceedings*, volume 3662 of *Lecture Notes in Artificial Intelligence*. Springer, 2005.

[11] Chitta Baral, Sarit Kraus, and Jack Minker. Combining multiple knowledge bases. *IEEE Transactions on Knowledge and Data Engineering*, 3(2):208–220, 1991.

[12] Peter Baumgartner, Peter Fröhlich, Ulrich Furbach, and Wolfgang Nejdl. Semantically guided theorem proving for diagnosis applications. In Martha E. Pollack, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence IJCAI 97, Volume 1*, pages 460–465, Nagoya, Japan, August 1997. Morgan Kaufmann.

[13] Rachel Ben-Eliyahu and Rina Dechter. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12(1–2):53–87, 1994.

[14] Gerhard Brewka. Logic programming with ordered disjunction. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, pages 100–105. AAAI Press, 2002.

[15] Antonio Brogi, Simone Contiero, and Franco Turini. Programming by combining general logic programs. *Journal of Logic and Computation*, 9(1):7–24, 1999.

[16] Antonio Brogi, Paolo Mancarella, Dino Pedreschi, and Franco Turini. Modular logic programming. *ACM Transactions on Programming Languages and Systems*, 16(4):1361–1398, 1994.

[17] Daniel R. Brooks, Esra Erdem, Selim T. Erdogan, James W. Minett, and Donald Ringe. Inferring phylogenetic trees using answer set programming. *Journal of Automated Reasoning*, 39(4):471–511, 2007.

[18] François Bry. A compositional semantics for logic programs and deductive databases. In Maher [89], pages 453–467.

[19] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Strong and weak constraints in disjunctive datalog. In Dix et al. [29], pages 2–17.

[20] Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in logic programming. *Journal of Logic Programming*, 19/20:443–502, 1994.

[21] Pedro Cabalar, David Pearce, and Agustín Valverde. Minimal logic programs. In Verónica Dahl and Ilkka Niemelä, editors, *Logic Programming, 23rd International Conference, ICLP 2007, Porto, Portugal, September 2007, Proceedings*, volume 4670 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2007.

[22] Marco Cadoli, Thomas Eiter, and Georg Gottlob. An efficient method for eliminating varying predicates from a circumscription. *Artificial Intelligence*, 54(2):397–410, 1992.

[23] Yin Chen, Fangzhen Lin, and Lei Li. SELP – a system for studying strong equivalence between logic programs. In Baral et al. [10], pages 442–446.

[24] Keith L. Clark. Negation as failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[25] Michael Codish, Saumya K. Debray, and Roberto Giacobazzi. Compositional analysis of modular logic programs. In *Conference Record of the Twentieth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 451–464, Charleston, South Carolina, United States, 1993. ACM Press.

[26] Johan de Kleer and Kurt Konolige. Eliminating the fixed predicates from a circumscription. *Artificial Intelligence*, 39(3):391–398, 1989.

[27] James P. Delgrande, Torsten Schaub, Hans Tompits, and Kewen Wang. A classification and survey of preference handling approaches in nonmonotonic reasoning. *Computational Intelligence*, 20(2):308–334, 2004.

[28] Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in DLV. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 847–852. Morgan Kaufmann, 2003.

[29] Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors. *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97, Dagstuhl Castle, Germany, July 1997, Proceedings*, volume 1265 of *Lecture Notes in Artificial Intelligence*. Springer, 1997.

[30] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative problem-solving using the DLV system. In Jack Minker, editor, *Logic-based artificial intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[31] Thomas Eiter, Wolfgang Faber, and Patrick Traxler. Testing strong equivalence of datalog programs - implementation and examples. In Baral et al. [10], pages 437–441.

[32] Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors. *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17–19, 2001, Proceedings*, volume 2173 of *Lecture Notes in Artificial Intelligence*. Springer, 2001.

[33] Thomas Eiter and Michael Fink. Uniform equivalence of logic programs under the stable model semantics. In Catuscia Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 2003.

[34] Thomas Eiter, Michael Fink, Hans Tompits, and Stefan Woltran. Simplifying logic programs under uniform and strong equivalence. In Lifschitz and Niemelä [77], pages 87–99.

[35] Thomas Eiter, Michael Fink, and Stefan Woltran. Semantical characterizations and complexity of equivalences in answer set programming. *ACM Transactions on Computational Logic*, 8(3):17, 2007.

[36] Thomas Eiter and Georg Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15(3–4):289–323, 1995.

[37] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive datalog. *ACM Transactions on Database Systems*, 22(3):364–418, 1997.

[38] Thomas Eiter, Georg Gottlob, and Helmut Veith. Modular logic programming and generalized quantifiers. In Dix et al. [29], pages 290–309.

[39] Thomas Eiter and Axel Polleres. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *Theory and Practice of Logic Programming*, 6(1–2):23–60, 2006.

[40] Thomas Eiter, Hans Tompits, and Stefan Woltran. On solution correspondences in answer-set programming. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, pages 97–102. Professional Book Center, 2005.

[41] Esra Erdem, Vladimir Lifschitz, and Donald Ringe. Temporal phylogenetic networks and logic programming. *Theory and Practice of Logic Programming*, 6(5):539–558, 2006.

[42] Esra Erdem, Vladimir Lifschitz, and Martin D. F. Wong. Wire routing and satisfiability planning. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic - CL 2000, First International Conference, London, UK, 24–28 July, 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 822–836. Springer, 2000.

[43] Sandro Etalle and Maurizio Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166(1–2):101–146, 1996.

[44] Sandro Etalle and Frank Teusink. A compositional semantics for normal open programs. In Maher [89], pages 468–482.

[45] Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic sets and their application to data integration. In Thomas Eiter and Leonid Libkin, editors, *Database Theory — ICDT 2005, 10th International Conference, Edinburgh, UK, January 2005, Proceedings*,

volume 3363 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2005.

[46] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In Alferes and Leite [2], pages 200–212.

[47] Paolo Ferraris. On modular translations and strong equivalence. In Baral et al. [10], pages 79–91.

[48] Haim Gaifman and Ehud Y. Shapiro. Fully abstract compositional semantics for logic programs. In *Conference Record of the Sixteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 134–142, Austin, Texas, USA, January 1989. ACM Press.

[49] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6–12, 2007*, pages 386–392. AAAI Press, 2007.

[50] Michael Gelfond. Representing knowledge in A-Prolog. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, volume 2408 of *Lecture Notes in Artificial Intelligence*, pages 413–451. Springer, 2002.

[51] Michael Gelfond and Alfredo Gabaldon. Building a knowledge base: an example. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):165–199, 1999.

[52] Michael Gelfond and Nicola Leone. Logic programming and knowledge representation — the A-Prolog perspective. *Artificial Intelligence*, 138(1-2):3–38, 2002.

[53] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Kowalski and Bowen [70], pages 1070–1080.

[54] Michael Gelfond and Vladimir Lifschitz. Compiling circumscriptive theories into logic programs. In Michael Reinfrank, Johan de Kleer, Matthew L. Ginsberg, and Erik Sandewall, editors, *Non-Monotonic Reasoning: 2nd International Workshop Grassau, FRG, June 1988, Proceedings*, volume 346 of *Lecture Notes in Artificial Intelligence*, pages 74–99. Springer, 1989.

[55] Michael Gelfond and Vladimir Lifschitz. Logic programs with classical negation. In David H. D. Warren and Péter Szeredi, editors, *Logic programming, Proceedings of the Seventh International Conference, Jerusalem, Israel, June 18–20, 1990*, pages 579–597. MIT Press, 1990.

[56] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–385, 1991.

[57] Laura Giordano and Alberto Martelli. Structuring logic programs: A modal approach. *Journal of Logic Programming*, 21(2):59–94, 1994.

[58] Keijo Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. *Fundamenta Informaticae*, 37(3):247–268, 1999.

[59] Maarit Hietalahti, Fabio Massacci, and Ilkka Niemelä. DES: a challenge problem for non-monotonic reasoning systems. In Chitta Baral and Mirosław Truszczyński, editors, *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, Breckenridge, Colorado, USA, April 2000. CoRR:cs.AI/0003039.

[60] Giovambattista Ianni, Giuseppe Ielpa, Adriana Pietramala, Maria Carmela Santoro, and Francesco Calimeri. Enhancing answer set programming with templates. In James P. Delgrande and Torsten Schaub, editors, *10th International Workshop on Non-Monotonic Reasoning (NMR 2004), Whistler, Canada, June 6–8, 2004, Proceedings*, pages 233–239, 2004.

[61] Katsumi Inoue and Chiaki Sakama. Negation as failure in the head. *Journal of Logic Programming*, 35(1):39–78, 1998.

[62] Katsumi Inoue and Chiaki Sakama. Equivalence of logic programs under updates. In Alferes and Leite [2], pages 174–186.

[63] Tomi Janhunen. On the effect of default negation on the expressiveness of disjunctive rules. In Eiter et al. [32], pages 93–106.

[64] Tomi Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16(1–2):35–86, 2006.

[65] Tomi Janhunen, Ilkka Niemelä, Dietmar Seipel, Patrik Simons, and Jia-Huai You. Unfolding partiality and disjunctions in stable model semantics. *ACM Transactions on Computational Logic*, 7(1):1–37, January 2006.

[66] Tomi Janhunen and Emilia Oikarinen. Testing the equivalence of logic programs under stable model semantics. In Sergio Flesca, Sergio Greco, Nicola Leone, and Giovambattista Ianni, editors, *Logics in Artificial Intelligence, 8th European Conference, JELIA 2002, Cosenza, Italy, September 2002, Proceedings*, volume 2424 of *Lecture Notes in Artificial Intelligence*, pages 493–504. Springer, 2002.

[67] Tomi Janhunen and Emilia Oikarinen. LPEQ and DLPEQ — translators for automated equivalence testing of logic programs. In Lifschitz and Niemelä [77], pages 336–340.

[68] David B. Kemp and Peter J. Stuckey. Semantics of logic programs with aggregates. In Vijay A. Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 – Nov. 1, 1991*, pages 387–401. MIT Press, 1991.

[69] Robert Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

[70] Robert A. Kowalski and Kenneth A. Bowen, editors. *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15–19, 1988.* MIT Press, 1988.

[71] Joohyung Lee and Fangzhen Lin. Loop formulas for circumscription. *Artificial Intelligence*, 170(2):160–185, 2006.

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

[73] Yuliya Lierler and Marco Maratea. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In Lifschitz and Niemelä [77], pages 346–350.

[74] Vladimir Lifschitz. Computing circumscription. In Aravind K. Joshi, editor, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence, 18–23 August 1985, Los Angeles, California*, pages 121–127. Morgan Kaufmann, 1985.

[75] Vladimir Lifschitz. Answer set planning. In Danny De Schreye, editor, *Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 – December 4, 1999*, pages 23–37. MIT Press, 1999.

[76] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1–2):39–54, 2002.

[77] Vladimir Lifschitz and Ilkka Niemelä, editors. *Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6–8, 2004, Proceedings*, volume 2923 of *Lecture Notes in Artificial Intelligence*. Springer, 2004.

[78] Vladimir Lifschitz, David Pearce, and Agustín Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.

[79] Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics in Artificial Intelligence*, 25(3–4):369–389, 1999.

[80] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *Logic Programming, Proceedings of the Eleventh International Conference on Logic Programming, June 13-18, 1994, Santa Margherita Ligure, Italy*, pages 23–37. MIT Press, 1994.

[81] Fangzhen Lin. Reducing strong equivalence of logic programs to entailment in classical propositional logic. In Dieter Fensel, Fausto Giunchiglia, Deborah L. McGuinness, and Mary-Anne Williams, editors, *Proceedings of the Eight International Conference on Principles and Knowledge Representation and Reasoning (KR-02), Toulouse, France, April 22–25, 2002*, pages 170–176. Morgan Kaufmann, 2002.

[82] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1–2):115–137, 2004.

[83] Zhijun Lin, Yuanlin Zhang, and Hector Hernandez. Fast SAT-based answer set solver. In *Proceedings, the Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 92–97, 2006.

[84] Per Lindström. First order predicate logic with generalized quantifiers. *Theoria*, 32:186–195, 1966.

[85] Lengning Liu and Mirosław Truszczyński. Pbmodels — software to compute stable models by pseudoboolean solvers. In Baral et al. [10], pages 410–415.

[86] John Wylie Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 1987.

[87] Michael J. Maher. Equivalences of logic programs. In Minker [96], pages 627–658.

[88] Michael J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110(2):377–403, 1993.

[89] Michael J. Maher, editor. *Logic Programing, Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming, September 2–6, 1996, Bonn, Germany*. MIT Press, 1996.

[90] Paolo Mancarella and Dino Pedreschi. An algebra of logic programs. In Kowalski and Bowen [70], pages 1006–1023.

[91] Victor W. Marek, Ilkka Niemelä, and Mirosław Truszczyński. Logic programs with monotone abstract constraint atoms. *Theory and Practice of Logic Programming*, 8(2):167–199, 2008.

[92] Victor W. Marek and Mirosław Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38(3):588–619, 1991.

[93] Victor W. Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In Krzysztof R. Apt, Victor W. Marek, Mirek Truszczyński, and David S. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer, 1999.

[94] John McCarthy. Circumscription — a form of non-monotonic reasoning. *Artificial Inteligence*, 13(1–2):27–39, 1980.

[95] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1–2):79–108, 1989.

[96] Jack Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1988.

[97] Andrzej Mostowski. On a generalization of quantifiers. *Fundamenta Mathematicae*, 44:12–36, 1957.

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

[99] Johannes Oetsch, Hans Tompits, and Stefan Woltran. Facts do not cease to exist because they are ignored: Relativised uniform equivalence with answer-set projection. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22–26, 2007, Vancouver, British Columbia, Canada*, pages 458–464. AAAI Press, 2007.

[100] Emilia Oikarinen. Modularity in SMODELS programs. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 2007, Proceedings*, volume 4483 of *Lecture Notes in Artificial Intelligence*, pages 321–326. Springer, 2007.

[101] Emilia Oikarinen and Tomi Janhunen. Verifying the equivalence of logic programs in the disjunctive case. In Lifschitz and Niemelä [77], pages 180–193.

[102] Emilia Oikarinen and Tomi Janhunen. Modular equivalence for normal logic programs. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 – September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, pages 412–416. IOS Press, 2006.

[103] Richard A. O'Keefe. Towards an algebra for constructing logic programs. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 152–160, Boston, Massachusetts, USA, July 1985. IEEE Computer Society Press.

[104] David Pearce, Vladimir Sarsakov, Torsten Schaub, Hans Tompits, and Stefan Woltran. A polynomial translation of logic programs with nested expressions into disjunctive logic programs: Preliminary report. In Peter J. Stuckey, editor, *Logic Programming, 18th International Conference, ICLP 2002, Copenhagen, Denmark, July/August 2002, Proceedings*, volume 2401 of *Lecture Notes in Computer Science*, pages 405–420. Springer, 2002.

[105] David Pearce, Hans Tompits, and Stefan Woltran. Encodings for equilibrium logic and logic programs with nested expressions. In Pavel Brazdil and Alípio Jorge, editors, *Progress in Artificial Intelligence, Knowledge Extraction, Multi-agent Systems, Logic Programming and Constraint Solving, 10th Portuguese Conference on Artificial Intelligence, EPIA 2001, Porto, Portugal, December 2001, Proceedings*, volume 2258 of *Lecture Notes in Artificial Intelligence*, pages 306–320. Springer, 2001.

[106] Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Partial stable models for logic programs with aggregates. In Lifschitz and Niemelä [77], pages 207–219.

[107] Teodor C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9(3–4):401–424, 1991.

[108] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1–2):81–132, 1980.

[109] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[110] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of constraint programming.* Series in Foundations of Artificial Intelligence. Elsevier, 2006.

[111] Yehoshua Sagiv. Optimizing datalog programs. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 23–25, 1987, San Diego, California*, pages 349–362. ACM Press, 1987.

[112] Chiaki Sakama and Katsumi Inoue. Embedding circumscriptive theories in general disjunctive programs. In Victor W. Marek, Anil Nerode, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning, Third International Conference, LPNMR'95, Lexington, KY, USA, June 1995, Proceedings*, volume 928 of *Lecture Notes in Artificial Intelligence*, pages 344–357. Springer, 1995.

[113] Chiaki Sakama and Katsumi Inoue. Prioritized logic programming and its application to commonsense reasoning. *Artificial Intelligence*, 123(1–2):185–222, 2000.

[114] Chiaki Sakama and Katsumi Inoue. Combining answer sets of nonmonotonic logic programs. In Francesca Toni and Paolo Torroni, editors, *Computational Logic in Multi-Agent Systems, 6th International*

Workshop, CLIMA VI, London, UK, June 27–29, 2005, Revised Selected and Invited Papers, volume 3900 of *Lecture Notes in Computer Science*, pages 320–339. Springer, 2006.

[115] Chiaki Sakama and Katsumi Inoue. Constructing consensus logic programs. In Germán Puebla, editor, *Logic-Based Program Synthesis and Transformation, 16th International Symposium, LOPSTR 2006, Venice, Italy, July 12–14, 2006, Revised Selected Papers*, volume 4407 of *Lecture Notes in Computer Science*, pages 26–42. Springer, 2007.

[116] Chiaki Sakama and Katsumi Inoue. Coordination in answer set programming. *ACM Transactions on Computational Logic*, 9(2):Article A9, 2008.

[117] Patrik Simons. Extending the stable model semantics with more expressive rules. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *Logic Programming and Nonmonotonic Reasoning, 5th International Conference, LPNMR'99, El Paso, Texas, USA, December 1999, Proceedings*, volume 1730 of *Lecture Notes in Artificial Intelligence*, pages 305–316. Springer, 1999.

[118] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.

[119] Timo Soininen and Ilkka Niemelä. Developing a declarative rule language for applications in product configuration. In Gopal Gupta, editor, *Practical Aspects of Declarative Languages, First International Workshop, PADL'99, San Antonio, Texas, USA, January 1999, Proceedings*, volume 1551 of *Lecture Notes in Computer Science*, pages 305–319. Springer, 1999.

[120] Timo Soininen, Ilkka Niemelä, Juha Tiihonen, and Reijo Sulonen. Representing configuration knowledge with weight constraint rules. In Alessandro Provetti and Tran Cao Son, editors, *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning, Papers from the AAAI Spring Symposium*, Stanford, USA, March 2001. AAAI Press. Technical Report SS-01-01.

[121] Tran Cao Son and Enrico Pontelli. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming*, 7(3):355–375, 2007.

[122] Robert F. Stärk. Input/output dependencies of normal logic programs. *Journal of Logic and Computation*, 4(3):249–262, 1994.

[123] Luis Tari, Chitta Baral, and Saadat Anwar. A language for modular answer set programming: Application to ACC tournament scheduling. In Marina De Vos and Alessandro Provetti, editors, *Answer Set Programming, Advances in Theory and Implementation, Proceedings of the 3rd International ASP'05 Workshop, Bath, UK, 27th–29th July, 2005*, volume 142 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.

[124] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[125] Hans Tompits and Stefan Woltran. Towards implementations for advanced equivalence checking in answer-set programming. In Maurizio Gabbrielli and Gopal Gupta, editors, *Logic Programming, 21st International Conference, ICLP 2005, Sitges, Spain, October 2005, Proceedings*, volume 3668 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2005.

[126] Hudson Turner. Strong equivalence made easy: Nested expressions and weight constraints. *Theory and Practice of Logic Programming*, 3(4–5):609–622, 2003.

[127] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[128] Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors. *Handbook of Knowledge Representation*. Elsevier Science, 2008.

[129] Sofie Verbaeten, Marc Denecker, and Danny de Schreye. Compositionality of normal open logic programs. *Journal of Logic Programming*, 42(3):151–183, 2000.

[130] Toshiko Wakaki and Katsumi Inoue. Compiling prioritized circumscription into answer set programming. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 2004, Proceedings*, volume 3132 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2004.

[131] Toshiko Wakaki and Kazuo Tomita. Compiling prioritized circumscription into general disjunctive programs. In Alessandro Provetti and Tran Cao Son, editors, *PREFS 2006: Preferences and their Applications in Logic Programming Systems, August 16th, 2006, Proceedings*, pages 1–15, Seattle, Washington, USA, 2006.

[132] Patrick Henry Winston. *Artificial Intelligence*. Addison-Wesley, 1992.

[133] Stefan Woltran. Characterizations for relativized notions of equivalence in answer set programming. In Alferes and Leite [2], pages 161–173.

[134] Stefan Woltran. A common view on strong, uniform, and other notions of equivalence in answer-set programming. *Theory Practice of Logic Programming*, 8(2):217–234, 2008.

TKK DISSERTATIONS IN INFORMATION AND COMPUTER SCIENCE

TKK-ICS-D1 Lehtimäki, Pasi
Data Analysis Methods for Cellular Network Performance Optimization. 2008.

TKK-ICS-D2 Laur, Sven
Cryptographic Protocol Design. 2008.

TKK-ICS-D3 Harva, Markus
Algorithms for Approximate Bayesian Inference with Applications to Astronomical Data Analysis. 2008.

TKK-ICS-D4 Ukkonen, Antti
Algorithms for Finding Orders and Analyzing Sets of Chains. 2008.

TKK-ICS-D5 Tatti, Nikolaj
Advances in Mining Binary Data: Itemsets as Summaries. 2008.

TKK-ICS-D6 Klami, Arto
Modeling of Mutual Dependencies. 2008.

ISBN 978-951-22-9581-4 (Print)
ISBN 978-951-22-9582-1 (Online)
ISSN 1797-5050 (Print)
ISSN 1797-5069 (Online)